STADIA
HELSINGIN AMMATTIKORKEAKOULU

TEKNIIKAN JA LIIKENTEEN TOIMIALA

Tietotekniikka

Ohjelmistotekniikka

INSINÖÖRITYÖ

MODULAARISEN TYÖASEMAOHJELMISTON TESTAUKSEN AUTOMATISOINTI

Työn tekijä: Jouni Pekkola
Työn valvoja: Auvo Häkkinen
Työn ohjaaja: Mika Salkola

Työ hyväksytty: __. __. 2007

Auvo Häkkinen
Yliopettaja

STADIA
HELSINGIN AMMATTIKORKEAKOULU

ALKULAUSE

Tämä insinöörityö on tehty eräälle yritykselle ja on osana insinöörikoulutustani Helsingin ammattikorkeakoulun tietotekniikkalinjan ohjelmistotekniikan koulutusohjelmassa. Työ suoritettiin yrityksen tiloissa 1.9.2006 ja 1.2.2007 välisenä aikana.

Työn valvojana toimi yliopettaja Auvo Häkkinen ja työnohjaajana toimi suunnittelupäällikkö Mika Salkola. Kiitän molempia annetuista suuntaviivoista ja lisäksi järjestelmäarkkitehtiä Mika Sormusta, joka antoi oman panoksensa insinöörityöhön aina, kun siinä esiintyi teknisiä ongelmia.

Järvenpää 25.1.2006

Jouni Pekkola

STADIA
HELSINGIN AMMATTIKORKEAKOULU

INSINÖÖRITYÖN TIIVISTELMÄ

Tekijä: Jouni Pekkola

Työn nimi: Modulaarisen työasemaohjelmiston testauksen automatisointi

Päivämäärä: 16.1.2007 Sivumäärä: 50 s. + 2 liitesivua

Koulutusohjelma: Tietotekniikka Suuntautumisvaihtoehto: Ohjelmistotekniikka

Työn valvoja: Auvo Häkkinen

Työn ohjaaja: Mika Salkola

Opinnäytetyön aiheena oli tutkia modulaarisen työasemaohjelmiston automatisoinnin mahdollisuuksia ja niiden toteuttamista käytännössä.

Teoriaosuudessa käsitellään ohjelmistotestauksen ja automatisoinnin peruskäsitteitä. Suurin osa työstä keskittyy testitapausten suunnitteluun, toteutukseen ja automatisointiin. Varsinaiset automatisointitoimet keskittyvät ohjelmiston laskennan toteuttavien moduulien testauksen automatisointiin. Testaustyökaluna käytettiin TestReflector nimistä apuohjelmaa, joka on yrityksen kehittämä työkalu ohjelmistokomponenttien testaukseen.

Jokaisen laskentatyypin testausta varten luotiin oma testiympäristö, joka kirjoitettiin C#-kielellä. Testiympäristön tärkein tehtävä oli alustaa tarvittavat tietorakenteet mittaustuloksilla ennen laskentojen suorittamista. Laskentojen testaukseen tarvittavat mittaustulokset saatiin laboratorioinstrumentilta. Mittaustulokset tuotiin testiympäristöön ohjelmiston käyttämästä tietokannasta XML-tiedostona.

Avainsanat: Atk-ohjelmat, testaus, automatisointi

STADIA
HELSINGIN AMMATTIKORKEAKOULU

ABSTRACT

Name: Jouni Pekkola

Title: Test automation of a modular workstation software

Date: 16.1.2007 Number of pages: 50 + 2

Department: Information technology Study Programme: Software engineering

Instructor: Auvo Häkkinen

Supervisor: Mika Salkola

The Theme of present Bachelor's Thesis was 'test automation of a modular workstation software'. The aim of the study was to investigate the possibilities of test automation and to deploy them in practice.

The theory section covers the basic principles of software testing and automation. The major part of the work is focused on the design, deployment and automation of testing the calculation modules of the software. A testing tool used was, named TestReflector, which was developed in-house for testing software components.

A decided test bed, written in C#, was created for each type of calculation. Its prime task was to initialize the software's data structures using measurement results prior to make calculations. The measurement results needed in the testing of the calculations were obtained from a laboratory instrument. The measurement results were exported from a database used by the software in XML format.

Keywords: Software, Testing, Automation

1 JOHDANTO

Ohjelmistotestauksen helpottamiseksi ja tehostamiseksi on kehitetty paljon erilaisia työkaluja ja menetelmiä. Yksi näistä menetelmistä on ohjelmistotestauksen automatisointi. Testauksen automatisoinnilla voidaan säästää jopa 80 % kustannuksissa. Pitää kuitenkin muistaa, että kertasuorituksena testauksen automatisointi voi tulla jopa huomattavasti kalliimmaksi. Automatisointi maksaa itsensä takaisin, kun sitä päästään tekemään toistuvasti. Automatisointi ei ole ratkaisu kaikkiin testauksen ongelmiin. /3, s.16–17. /

Tämän insinöörityön tarkoituksena on tutkia erään yrityksen kehittämän ohjelmiston testauksen automatisointimahdollisuuksia. Teoriaosuuden luvuissa 2 ja 3 annetaan lukijalle tarvittavat tiedot ohjelmistotestauksesta ja testauksen automatisoinnista, jotta hän pystyy ymmärtämään käytännön osuudessa kuvattua toteutusta. Luvussa 4 esitellään yrityksen tuotteita ja yrityksen ohjelmistotuotantoa ja testauskäytäntöä. Varsinainen testauksen automatisointiprosessin kuvaus aloitetaan luvussa 5, jossa käydään läpi yrityksen nykyinen testausjärjestelmä ja valitaan automatisoitavat kohteet. Automatisoinnin kohteeksi valittiin ohjelmiston laskentoja suorittavat moduulit. Luvussa 6 esitellään, minkälaisia laskentoja ohjelmisto voi suorittaa. TestReflector-niminen työkalu esitellään (luku 7) ennen varsinaisen testien suunnittelun ja toteutuksen aloittamista, joka on tämän työn tärkein osa-alue.

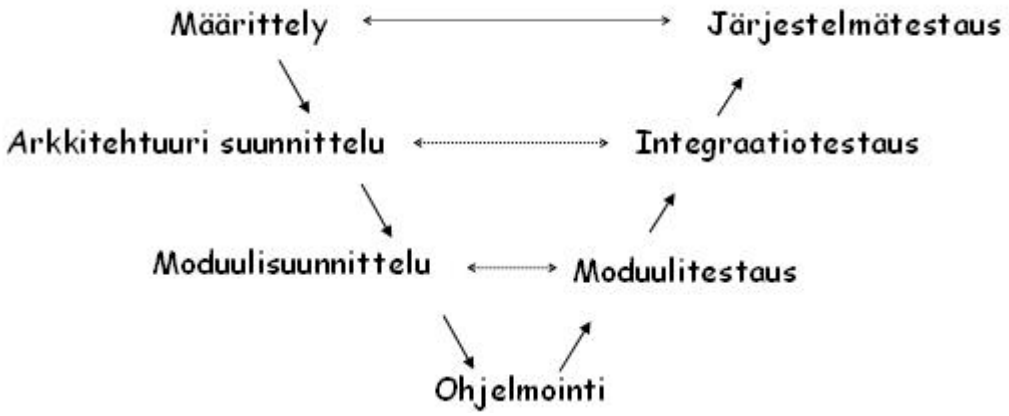
Työn tärkeimmiksi tavoitteiksi on määritelty testaustyön tehostaminen ja testauksessa syntyvien kustannusten alentaminen. Nykyisessä järjestelmässä yrityksen testaajat testaavat ohjelmaversiosta toiseen samoja testitapauksia, vaikka testitapausten kohteena olevassa osassa ei ole tapahtunut mitään muutoksia. Kun tällaiset testitapaukset automatisoidaan, voidaan testausresursseja vapauttaa paljon kriittisempään testaamiseen. Automatisoinnista syntyy ajan myötä myös kustannussäästöjä. Tarkastellaan automatisoinnista syntyviä säästöjä kuvitteellisten lukujen valossa. Oletetaan, että yhden ohjelmaversion testaukseen käytetään 650 tuntia ja se tulee maksamaan 39 000 €. Jos testaamiseen käytettyä aikaa pystytään pienentämään esimerkiksi 10 % automatisoinnin avulla, säästetään työtunteja 65 ja rahaa säästyy 3 900 €. Lisäksi ohjelmaversion läpimeno testausprosessissa nopeutuu noin kaksi viikkoa.

2 OHJELMISTOTESTAUS

Luku esittelee ohjelmistotestauksen keskeisimpiä asioita. Käydään läpi testauksen tasot ja esitellään keskeisimmät testauksen menetelmät.

2.1 Testauksen tasot

Testauksen V-malli pitää sisällään erilaisia testauksen tasoja, joita ovat moduulitestaus, integraatiotestaus ja järjestelmätestaus (kuva 1). Järjestelmätestausta voi seurata vielä hyväksyttämistestaus.



Kuva 1. Testauksen V-malli /2, s. 287./

Moduulitestauksessa tarkoituksena on testata ohjelmistoon kuuluvan yksittäisen moduulin toimivuutta. Moduulitestauksessa voidaan joutua rakentamaan ”testipetejä” (engl. testbed), jotta moduulin toimivuus saadaan varmistettua. Testiohjelmat sisältävät ohjelmiston toimintaa simuloivia osia, jotka ovat useimmiten tynkämoduuleja (engl. test stubs) tai ajureita. /2, s. 288./

Integraatiotestaukseen kuuluu moduulien tai moduuliryhmien yhdistäminen. Testauksen pääpaino on rajapintojen toiminnan tutkimisessa. Integraatiotestausta tehdään usein rinnakkain moduulitestauksen kanssa. Integraatiotestaus etenee usein kokoavasti (engl. bottom-up), eli alemman tason moduuleista ylöspäin. Tuloksia verrataan vaatimusmäärittelyihin. /2, s. 289./

Järjestelmätestauksessa tarkastellaan koko järjestelmän toimivuutta ja verrataan tuloksia määrittelydokumenttiin ja asiakasdokumenttiin. Testaushenkilöksi tulisi valita testauksesta mahdollisimman riippumaton henkilö. Järjestelmätestauksessa testataan myös ei-toiminnalliset ominaisuudet. /2, s. 290./

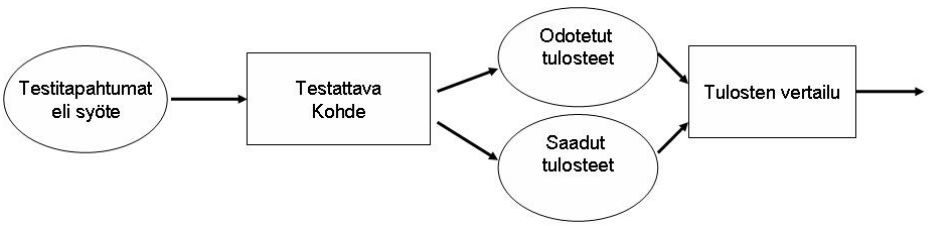
Hyväksyttämistestauksessa asiakas varmistaa, että hänen asettamansa vaatimukset ohjelman toiminnallisuudesta ovat täyttyneet. Kun ohjelmiston testaus suoritetaan toimittajan tiloissa, puhutaan alfa-testauksesta. Beta-testauksessa asiakas tai ulkopuoliset testaajat hoitavat testauksen itsenäisesti omissa tiloissaan. /2, s. 290./

2.2 Testauksen menetelmät

Kun puhutaan testauksen menetelmistä, tarkoitetaan yleensä lasilaatikko- (engl. whitebox), mustalaatikko- (engl. blackbox) tai harmaalaatikkomenetelmää (engl. greybox).

2.2.1 Lasilaatikko

Lasilaatikkotestauksella tarkoitetaan menetelmää, jossa testitapaukset johdetaan esimerkiksi kohteen sisäisestä rakenteesta tai logiikasta. Tavoitteena on valita käytäntö, jossa ohjelmiston kaikki osat tulisi käytyä mahdollisimman hyvin läpi. Testauksen kattavuutta voidaan tarkastella lausekattavuuden avulla. Lausekattavuudessa testitapauksien tulisi kattaa kaikki ohjelmiston lauseet. Huono puoli tässä menetelmässä on se, että ohjelmistojen koon kasvaessa myös testitapausten määrät kasvavat liian suuriksi. Toinen käytetty menetelmä on haarojen kattavuus. Tämän menetelmän mukaan testauksen tulisi kattaa kaikki ohjelmiston mahdolliset haarat. Tämän menetelmän huono puoli on se, että se soveltuu oikeastaan ainoastaan funktioiden testaamiseen, jossa eri polkuvaihtoehtojen määrä ei kasva liian suureksi. /2, s. 291/

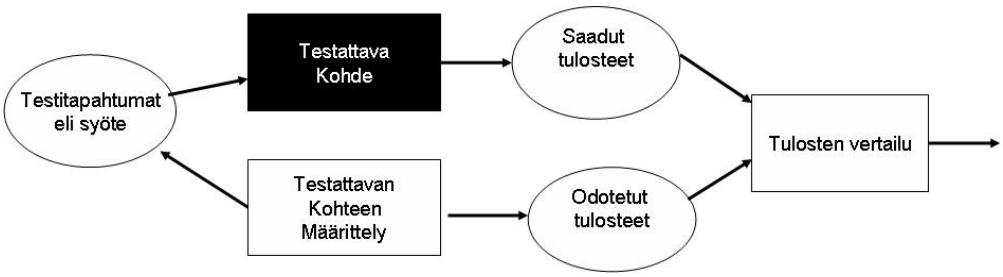


Kuva 2. Lasilaatikkomenetelmä /7, s. 16. /

Kuvassa 2 on esitelty lasilaatikkomenetelmä kaaviokuvana. Kuvassa testitapahtuma syötetään testattavalle kohteelle, joka esitellään lasilaatikkona. Testattavan kohteen toteutus tunnetaan. Testattavan toteutuksen pohjalta saadaan selville odotetut ja saadut tulokset. Odotettuja ja saatuja tuloksia vertaamalla voidaan päätellä, onko testi onnistunut vai ei.

2.2.2 Mustalaatikko

Mustalaatikkotestauksessa testitapaukset luodaan yleensä ohjelmistojen määrittelyjen pohjalta. Testauksessa otetaan huomioon ohjelmistolle annetut syötteet ja ohjelmiston palauttamat arvot. Testitapausten valitseminen perustuu erilaisiin menetelmiin. Yksi menetelmä on syöteavaruuden jakaminen ekvalenssiluokkiin. Oletuksena voidaan sanoa, että jos ohjelmisto toimii yhdellä syötteen arvolla, niin se toimii myös muilla. Tällä tavalla jaetusta aineistosta käytetään nimitystä ekvalenssiositus. Esimerkiksi syöteavaruus voidaan jakaa ekvalenssiluokkiin seuraavasti. Ensimmäisen luokat muodostavat epäkelvolliset syötteet. Toisen luokan muodostavat pienet tai suuret syötteet ja kolmannen luokan muodostavat kaikki kelvolliset syötteet. Yksi hyvin käytetty menetelmä on raja-arvoanalyysi. Raja-arvoanalyysin perusolettamuksena on ajatus, että jos testi epäonnistuu jollakin syötteen arvolla, se epäonnistuu myös kaikilla muilla arvoilla, jotka ovat raja-arvon sisällä. /2, s.29./



Kuva 3. Mustalaatikkomenetelmä /7, s. 20./

Kuvassa 3 on kuvattu mustalaatikkotestauksen eteneminen. Testaus käynnistyy testattavan kohteen määrittelystä, josta johdetaan testitapahtumat ja odotetut tulokset. Testattava kohde kuvataan mustana laatikkona, eli kohteen toteutusta ei tunneta. Testattava kohde antaa tulokset testattavan testisyötteen pohjalta. Tulosten vertailu tehdään kuten lasilaatikkomenetelmässä saadun ja odotettujen tulosten välillä.

2.2.3 Harmaalaatikko

Harmaalaatikkotestaus on lasilaatikko- ja mustalaatikkotestauksen välimuoto. Testitapausten etsinnässä käytetään tietoa ohjelman toteutusperiaatteista. Harmaalaatikkotestausta voidaan hyödyntää ohjelmiston kriittisten osien testauksessa. Voidaan valita tietoisesti vaikeita tapauksia ja vääriä syötteitä sekä testata, miten ohjelmisto toipuu virhetilanteista. /8, s. 20–21/

3 TESTAUKSEN AUTOMATISOINTI

Luvussa tutustutaan käsitteeseen testauksen automatisointi. Siinä esitellään edut ja haitat, joita automatisointi tuo tullessaan. Luvun lopussa esitellään menetelmiä, joilla automatisointia voidaan toteuttaa.

3.1 Ohjelmistotestauksen automatisointi

”Automatisointi on manuaalitestauksen suorittamista koneellisesti, jonkin testausohjelman avulla. Kuitenkin ihminen on silti tärkein osa”, määrittelee Fester testauksen automatisoinnin. /3, s. 5./

Ennen kuin voidaan automatisoida yhtään mitään, pitää olla toimiva manuaalitestausjärjestelmä. Manuaalitestausjärjestelmän pitää täyttää seuraavat kriteerit: selkeät yksityiskohtaiset testitapaukset ja odotettavissa olevat tulokset, jotka perustuvat vaatimusmäärittelyihin.

Testauksen automatisoinnin voima perustuu toistettavuuteen (engl. repeatability), laajennettavuuteen (engl. leverage) ja kertymään (engl. accumalation).

Toistettavuudella tarkoitetaan, että testit voidaan suorittaa useita kertoja samanlaisina. Laajennettavuuskäsite tarkoittaa, että voidaan suorittaa sellaisia testejä, jotka ovat manuaalisesti suoritettuina mahdottomia. Kertymä tarkoittaa mahdollisuutta suoriutua sovelluksen muutoksista vähemmällä määrällä testejä.

Kun testausta lähdetään automatisoimaan, automatisoidaan jotakin seuraavia työvaiheita: testauksen ehtojen tunnistamista, testitapausten suunnittelua, testien rakentamista sekä testien suorittamista tai tulosten vertailua olemassa oleviin tuloksiin.

3.2 Automatisoinnin edut

Suurimmat edut automatisoinnista saavutetaan silloin, kun olemassa olevia testejä voidaan suorittaa uudestaan uudella ohjelmaversiolla. Tämä etu on suuri varsinkin ympäristössä, jossa ohjelmia versioidaan säännöllisesti. Automatisoinnin tärkein etu on se, että pystytään suorittamaan suuren määrä testejä nopeasti. Automatisoidut testit toistetaan aina samassa

muodossa, mikä antaa luotettavuutta, jota manuaalitestauksella ei voida saada aikaan. Hyvin suunniteltu automatisointi mahdollistaa resurssien vapauttamisen muuhun kriittisempään testaustyöhön. /3, s. 9./

3.3 Automatisoinnin ongelmat

Yleinen harhaluulo on, että ohjelmistotestauksen automatisointi on ratkaisu kaikkiin testauksen ongelmiin, mutta sitä se ei ole. Automatisointiprojektit voivat epäonnistua yhtä useasti kuin mikä tahansa muukin ohjelmistoprojekti. Suurimmat syyt automatisoinnin epäonnistumiseen ovat epärealistiset odotukset, ylläpidettävyyden vaikeudet ja työkalut.

Kun ohjelmisto muuttuu, pitää päivittää testejä tai jopa kaikki testit. Ylläpidettävyyden ongelma on lopettanut monta automatisointiprojektia heti alkuunsa, kun havaitaan testien ylläpidon olevan liian työlästä tai mahdotonta.

Työkalu itsessään voi aiheuttaa myös ongelmia. Testiautomaatiotyökalut ovat myös ohjelmistoja ja voivat sisältää myös virheitä. On turhauttavaa havaita, että työkalu on huonosti testattu ja sisältää suuren määrän virheitä. /1, s. 10–13./

4 YRITYKSEN OHJELMISTOKEHITYS

Tämä luku käsittelee testauksen automatisoinnin kohteena olevaa ohjelmistoa ja esittelee yrityksen ohjelmistotuotannon menetelmiä ja kehitysympäristön. Luvun lopussa esitellään yrityksen nykyistä testausjärjestelmää.

4.1 Mittalaitteet

Yrityksen valmistamista tuotteista ovat laboratoriomittalaiteita. Mittalaitteet tekevät pääsääntöisesti fotometrisiä, fluorometrisiä ja luminometrisiä mittauksia.

Fotometrisessä mittaustekniikassa syötetään tiettyä aallonpituutta näytteen läpi. Mittauksen kohteena ovat näytteen läpäisevän valon aallonpituudet ja niiden intensiteetit. /6, s.136–140./

Fluorometrisessä mittauksessa näyte altistetaan jollekin ennalta määrätylle valon aallonpituudelle. Tämän altistamisen aikana näytteet virittyvät uudelle energiatasolle. Kun virittävä valo sammutetaan, näytteet palautuvat takaisin omalle energiatasolle ja emittoivat toista aallonpituutta. Tämän emittoituneen aallonpituuden intensiteetti mitataan. Fluorometrisiin mittauksiin kuuluu myös aikaerotteinen fluorometrinen mittaus. Idea on sama kuin normaalissa fluorometrisessä mittauksessa, mutta aikaerotteisessa mittauksessa tarkastellaan lisäksi mittaussignaalin intensiteettiä ajan funktiona. /6, s.130-135/

Luminometrisessä mittauksessa mitataan näytteistä tiettyjen kemiallisten tai biokemiallisten reaktioiden johdosta vapautuvan valon aallonpituutta ja intensiteettiä. /6, s. 127–130./

4.2 Ohjelmiston esittely

Työn kohteena oleva ohjelmisto on laboratoriomittalaitteiden työasemaohjelmisto. Ohjelmiston avulla pystytään määrittämään halutut mittaukset, näytteiden tyypit ja määrät. Mittaustuloksista voidaan suorittaa useita erilaisia laskentoja ja tulostaa kattavia raportteja.

Työasemaohjelmisto tarvitsee toimiakseen Microsoft .NET-sovelluskehyksen ja Microsoft SQL-tietokantapalvelimen. Ohjelmisto tallentaa kaiken tiedon

SQL-palvelimeen, josta esimerkiksi kaikki mittaustulokset ovat saatavissa laskentojen suorittamista varten.

4.3 Kehitysympäristö

Juuri nyt yrityksen ohjelmistokehitysympäristö on muutoksen kohteena. Tällä hetkellä ohjelmistokehityksen vaatimustenhallintaa ja testiympäristöä ylläpitää alihankkijayritys. Ohjelmistojen vaatimustenhallintaa ja sen ylläpitoa ollaan siirtämässä yritykselle itselleen. Toinen merkittävä muutos tulee olemaan, jos yrityksessä siirrytään vanhasta Rationalin tuotteiden ympärille rakennetusta ympäristöstä uuteen Microsoftin tarjoamaan Team Foundation Serveriin. Eräs hyvin tärkeä syy ympäristön vaihtamiseen on se, että Team Foundation Server on huomattavasti kevyempi ja nykyaikaisempi kuin Rationalin vastaavat järjestelmät. Toinen siirtymiseen vaikuttava syy on se, että Team Foundation Server tarjoaa tärkeimmät työkalut koko ohjelmistoprojektien elinkaaren hallinnoimiseen samassa paketissa. Oikeastaan ainoa heikkous Rationalin järjestelmään verrattuna on se, että Team Foundation Server ei sisällä kehittyneitä työkaluja testitapauksien hallinnoimiseen. Rationalin järjestelmässä on erinomaiset työkalut tätä varten. /8;10/

4.4 Testauskäytännön esittely

Yrityksen ohjelmiston järjestelmätason testauksen suorittaa suurimmilta osin alihankkijayrityksen edustajat. Jos asiaa ajatellaan V-mallin tasojen kannalta, moduuli- ja integraatio-testaamisesta vastaa yrityksen sovelluskehittäjät ja omat testiresurssit. Moduulitason testauksessa ideana on se, että tietyn sovelluksen osa-alueen kehittäjä testaa itse tekemänsä koodin toiminnasta. Tämä järjestely on ohjelmiston kannalta toiminut erittäin hyvin kahdesta syystä. Ensinnäkin alihankkijalla on käytettävissä huomattavasti paremmat resurssit ohjelmiston testaamiseen kuin yrityksellä itsellään. Alihankkijalla on mahdollisuus asettaa resursseja testaamaan ohjelmistoa täysipäiväisesti. Toiseksi järjestelmätason testauksessa ulkopuoliset testaajat tuovat testaamiseen uudenlaisen riippumattoman näkökulman. Voisi ajatella, että heidän työnsä tuloksen mittari on se, miten paljon he löytävät ohjelmistosta virheitä. /8/

Ohjelmiston testaamiseen käytetään mustalaatikkomenetelmää. Testaaminen toteutetaan manuaalisesti. Mustalaatikkomenetelmän mukaisesti ohjelmistoon liittyvät testitapaukset on laadittu ohjelmiston vaatimusten pohjalta, ja osa testitapauksista on kirjoitettu jopa ennen kuin

varsinaista koodia on kirjoitettu riviäkään. Testitapauksista on laadittu hyvät testaussuunnitelmat. Suunnitelmat sisältävät tiedon, miten järjestelmän tulisi toimia, jotta se vastaisi spesifikaatioita. /8./

5 AUTOMATISOITAVIEN TESTITAPAUSTEN KARTOITTAMINEN

Ennen varsinaisen automatisointiprosessin aloittamista käytiin läpi ne valintakriteerit, joiden avulla olemassa olevien testitapausten joukosta valitaan automatisoitavat testit. Tässä luvussa esitellään myös miten testaukseen käytetty aika jakautuu eri testitapausten kesken. Luvussa valitaan ne testitapaukset, joiden testaaminen tullaan automatisoimaan.

5.1 Valintakriteerit

Ennen kuin aloitetaan automatisoiminen, täytyy määritellä kriteerit, joiden avulla testattavat kohteet valitaan. Ensimmäinen tärkeä kriteeri on se, että testeillä on selkeä hyvin laadittu testaussuunnitelma. Hyvä testaussuunnitelma toimii hyvänä ohjenuorana, kun automatisointia suunnitellaan.
Toinen tärkeä kriteeri on testitapauksen manuaalisesti suorittamiseen käytetty aika. Testitapauksien automatisoinnin toteuttaminen on kallista ja vaikeaa. Ei ole kustannustehokasta ryhtyä automatisoimaan testejä, joiden suorittamiseen menee testaajalta vain vähän aikaa.

Kolmas tärkeä kriteeri on testien ylläpidettävyys. Monet automatisointiprojektit ovat kaatuneet ylläpidettävyyden ongelmaan. Tarkoituksena on valita sellaisia testitapauksia, jotka ovat nyt hyvin stabiilissa muodossa eivätkä enää muutu kovin dramaattisesti, vaikka ohjelmistossa tapahtuisikin jotakin muutoksia. Stabiilien testitapausten löytäminen helpottaa huomattavasti ylläpidettävyysongelmaa.

Viimeisenä kriteerinä testitapauksilla tulisi olla selkeät kelvolliset ja epäkelvolliset syöteavaruuden arvot. Kun voidaan määrittää syöteavaruudet, voidaan määrittää syöteavaruuden arvoille vertailuarvot. Näin voidaan varmistaa testikohteen oikea toiminnallisuus. Epäkelvollisen syöteavaruuden arvoilla voidaan testata miten ohjelmisto reagoi, kun ohjelmistoon syötetään arvoja, joiden ei pitäisi olla mahdollisia.

5.2 Testauksen aikajakauma ja testitapauksien esittely

Yrityksellä on hyvin kattava tilasto siitä, kuinka paljon testaajilta on kulunut aikaa tietyn testitapauksen suorittamiseen. Tämän tilasto tarjoaa hyvän apuvälineen, kun lähdetään miettimään mitä kannattaa automatisoida.

Tässä kappaleessa esitellään testitapaukset joista ohjelmiston testaus koostuu, ja kappaleen lopussa tarkastellaan ajan jakautumista eri testitapausten kesken.

5.2.1 ARB-testit

ARB-lyhenne tulee sanoista Archiving, Restore ja Backup. Testien kohteena on ohjelmiston tietokanta. Testit on jaettu kolmeen ryhmään: arkistointiin (engl. archiving), palauttamiseen (engl. restore) ja varmistamiseen (engl. backup).

Arkistointiin liittyvät testit käsittelevät pääsääntöisesti tiedon hallinnoimiseen liittyviä asioita. Testit testaavat esimerkiksi kirjautumista tietokantaan, tiedon hakemista ja käsittelyä tietokannasta.

Palautustesteissä testataan tietokannan palautusta varmistuksesta mahdollisen vikatilanteen syntyessä.

Varmistukseen liittyvät testit käsittelevät toimivan varmistuksen tekemistä käytössä olevasta tietokannasta. Testitapaukset käsittelevät esimerkiksi varmistettavaan tietokantaan kirjautumista tai varmistuksen tekemistä sisäisen verkon verkkolevylle tai paikalliselle kovalevylle.

5.2.2 Valikoiva testaaminen

Valikoivassa testauksessa (engl. explorative testing) ei ole suunniteltu valmiiksi mitään erityistä testaussuunnitelmaa. Testaajat toimivat intuition avulla. Testaajat pyrkivät kuormittamaan testaamisella niitä ohjelman osia, joissa kuvittelevat ongelman sijaitsevan.

5.2.3 Sovelluskehyksen uudistukset

Sovelluskehyksen uudistukset ryhmää kuuluvien testien tehtävänä on varmistaa ohjelmiston toiminta Microsoft .NET 2.0 sovelluskehyksellä. Testit koostuvat pääsääntöisesti vanhoista testitapauksista, joita suoritetaan vain .NET 2.0-alustalla.

5.2.4 Laskentatestit

Yleisessä läpikäynnissä (engl. Smoketest) testataan uuden ohjelmaversion toimivuutta. Yleisen läpikäynnin kohteita ovat esimerkiksi uuden ohjelmaversion asennus ja perustoiminnallisuuden toteaminen, kuten laskennat, käyttäjien lisääminen ja mittausprotokollien luonti.

5.2.5 *Yleinen läpikäynti*

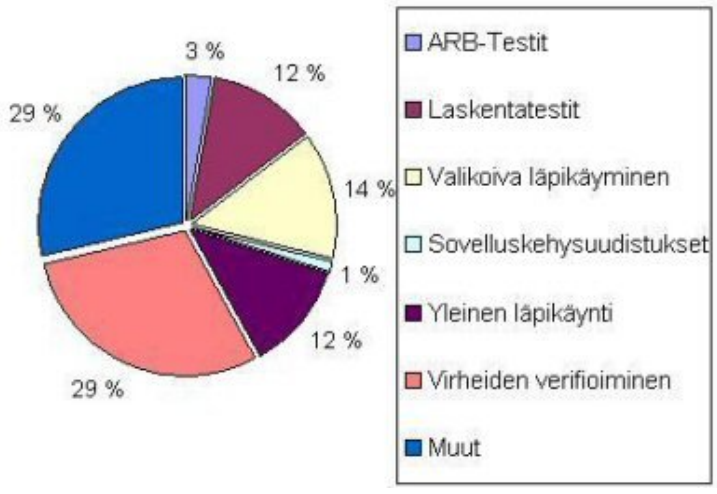
Yleisessä läpikäynnissä (engl. Smoketest) testataan uuden ohjelmaversion toimivuutta. Yleisen läpikäynnin kohteita ovat esimerkiksi uuden ohjelmaversion asennus ja perustoiminnallisuuksien toimiminen, kuten laskennat, käyttäjien lisääminen ja mittausprotokollien luonti.

5.2.6 *Virheiden verifioiminen*

Virheiden verifioimisella tarkoitetaan oikeastaan ”virheiden elinkaarta”. Sen tarkoituksena on verifioida, että kehittäjien korjatuiksi merkitsemät virheet ovat todella tulleet kuntoon.

5.2.7 *Testauksen aikajakauma*

Kuva 4 esittelee kuinka eri testitapauksien testaustunnit ovat jakautuneet suhteessa kokonaistestaustuntimäärään. Kuvan 4 materiaali on kerätty yrityksen aineistosta, jossa seurataan miten paljon eri kohteiden testaamiseen on käytetty aikaa. Käytettävissä olevasta aineistosta on nostettu esille ne testitapaukset, joiden suorittamiseen on kulunut 20 tuntia tai yli. Kaikki muut testitapaukset kuuluvat ryhmän ”muut” alle. Yli 20 tuntia vievien testitapausten joukosta valittiin ne testitapaukset, joiden automatisointi on mielekästä ja mahdollista.



Kuva 4. Testauksen aikajakauma

Kuten kuvasta 4 voidaan päätellä suurimman viipaleet kuvaajasta vievät virheiden verifioiminen ja muut 29 %. Seuraavan merkittävän osuuden kuvaajasta vie valikoiva läpikäyminen 14 % osuudella. Kolmannella sijalla ovat laskentatestit ja yleinen läpikäynti 12 % osuudella. Vähiten kuvaajasta vievät ARB-testit 3 % osuudella ja sovelluskehyksen uudistukset 1 % osuudella.

5.3 Automatisoitavien testitapausten valinta

Edellisessä luvussa esiteltiin, miten testauksessa käytettävä aika jakautuu eri testitapausten välille. Ryhmä ”muut” hylättiin heti, koska tämän ryhmän testitapaukset ovat sellaisia testejä, joiden suorittamiseen testaajalta on kulunut hyvin vähän aikaa. Työn tavoitteiden kannalta tällaisten testien automatisointi ei ole tarpeellista.

Testejä tarkasteltiin kohdassa 5.1 asetettujen mukaisesti. Ensimmäisenä vaadittiin, että testeillä piti olla selkeät testaussuunnitelmat. Selkeät testaussuunnitelmat löytyivät ARB-, laskenta- ja yleisiltä läpikäyntitesteiltä. Yrityksen tietokannasta ei löytynyt mitään virallista testaussuunnitelmaa sovelluskehysuudistuksille, virheiden verifioimiselle ja valikoivalle testaamiselle, joten nämä vaihtoehdot jouduttiin hylkäämään.

Toisena kriteerinä oli testaamisen käytetty aika. Tämän kriteerin näkökulmasta tarkasteltuna ARB-testit ja sovelluskehysuudistukset piti pudottaa pois, koska ARB-testit vievät vain 3 % ja sovelluskehysuudistukset 1 % kaikesta testaukseen käytetystä ajasta. Laskentatestit ja yleinen läpikäynti vievät kumpikin 12 % kaikesta testaukseen käytetystä ajasta.

Kolmantena kriteerinä määriteltiin testitapausten ylläpidettävyys. Testauksen kohteeksi valittiin sellaiset ohjelman osa-alueet, jotka ovat mahdollisimman stabiilissa muodossa, eivätkä välttämättä muutu merkittävästi vaikka ohjelmisto kehittyisikin eteenpäin. Tällaisia testitapauksia ovat ryhmään *laskentatestit* kuuluvat testitapaukset. Tietyn laskutoimituksen suorittamiseen liittyvä logiikka pysyy samana, vaikka ohjelmaan tulisikin muutoksia. Tämä helpottaa huomattavasti testien ylläpidettävyyttä testauksen automatisoinnissa.
Laskentatestiryhmään kuuluvat testit täyttävät myös neljännen kriteerin. Laskentatesteille on mahdollista määrittää kelvolliset ja epäkelvot syöteavaruudet. Näille syötearvolle pystytään laskemaan myös oikea tulos etukäteen.

6 OHJELMISTON LASKENNAT

Tässä luvussa tutustutaan laskentoihin, joita ohjelmassa voidaan suorittaa. Tarkoituksena on antaa yleiskuva kaikista ohjelmassa esiintyvistä laskentatyypeistä.

6.1 Perustilastotiedot

Perustilastotietojen laskennoissa lasketetaan keskiarvoa, keskihajontaa ja variaatiokerrointa. Keskiarvomittauksella tarkoitetaan jonkin näytteen mittaustuloksen keskiarvoa.

Keskihajonta on hajontaluku välimatka- tai suhdeasteikon mittaustuloksille. Keskihajonta on ehkä kaikkein yleisimmin käytetty hajontaluku. Keskihajonta kuvaa sitä, kuinka kaukana yksittäiset mittaustulokset ovat keskimäärin mitatusta aritmeettisesta keskiarvosta. Keskihajonta lasketaan kaavasta

$$S = \sqrt{\frac{\frac{1}{n}\sum_{n-1}(x_i - \bar{x})^2}{n-1}} \qquad (1)$$

Kaavassa x_i viittaa i:nnen havainnon arvoon ja $\bar{x}$ tarkoittaa aineiston aritmeettista keskiarvoa. Sigma – merkki (Σ) tarkoittaa summaa. Esitetyssä kaavassa lasketaan jokaisen havainnon arvon erotus koko aineiston keskiarvosta. Tämän jälkeen erotus korotetaan neliöön. Tämän jälkeen kaikki saadut arvot lasketaan yhteen. Tämä saatu summa jaetaan havaintojen määrällä (n) ja saadusta tuloksesta otetaan vielä neliöjuuri keskihajonnan saamiseksi. Mitä suurempi saatu arvo on, sitä enemmän muuttujan arvoissa on hajontaa ja päinvastoin.

Kahden eri otoksen keskihajontojen keskinäinen vertailu on joskus ongelmallista, koska keskihajonta vaihtelee aineiston keskiarvon myötä. Variaatiokerroin on hajontaluku, joka suhteuttaa keskihajonnan aineiston keskiarvoon. Variaatiokerroin lasketaan kaavasta

$$V = \frac{S}{X} * 100 \qquad (2)$$

Kaavassa s on muuttujan keskihajonta ja $\bar{x}$ on muuttujan keskiarvo. Käytännössä siis keskihajonta suhteutetaan muuttujan keskiarvoon. Näin kahden ryhmän hajonnan vertailu on mielekkäämpää.

6.2 Taustakohinan vähentäminen

Taustakohinan vähentämistä käytetään poistamaan ympäristön vaikutus mittaustuloksesta. Ohjelmassa voidaan määritellä useampi ”tyhjä näyte” ja laskea näiden kaikkien mittausten keskiarvo. Tämä keskiarvo vähennetään kaikkien muista todellisten näytteiden mittaustuloksista. Toinen tapa taustakohinan vaikutuksen vähentämiseksi mittaustuloksesta on se, että jokaiselle mittaustulokselle määritetään oma ”tyhjä näytteenä”. Tämä tyhjän näytteen arvo vähennetään sille osoitetusta mittaustuloksesta. /4, s.109/

6.3 Tiedon normalisointi

Tiedon normalisoinnissa selvitetään suureita suhdelukua (engl. ratio) ja estoluku (engl. inhibitio). Suhdeluvulla tarkoitetaan aina kontrollinäytteen ja jonkin toisen näytteen välistä suhdetta. Kaavassa B on jokin ennalta määrätty näyte ja B_0 kontrollinäyte. /4, s. 110./

$$Ratio = \frac{B}{B_0} * 100\% \qquad (3)$$

Hyvä esimerkki ko. suhdeluvun käytöstä on kemiallisentutkimus. Suhdelukua käytetään silloin, kun tarkastellaan esimerkiksi jonkin kemiallisen yhdisteen vaikutusta biologiseen organismiin. Kaavan avulla saadaan esimerkiksi selvitettyä kuinka nopeasti yhdisteen vaikutus häviää organismista.

Estoluku on tavallaan käänteinen toimenpide suhdeluvulle. Jos esimerkiksi suhdeluvun kohdalla tarkasteltiin lääkeaineen vaikutuksen vähenemistä, estoluvulla tarkastellaan jokin aineen vähenemisen estäviä tekijöitä. Estoluku lasketaan kaavasta, jossa B on jokin ennalta määrätty näyte ja B0 kontrollinäyte.

$$Inhibitio = 100 - \frac{B}{B_0} * 100\% \qquad (4)$$

6.4 Kineettiset laskennat

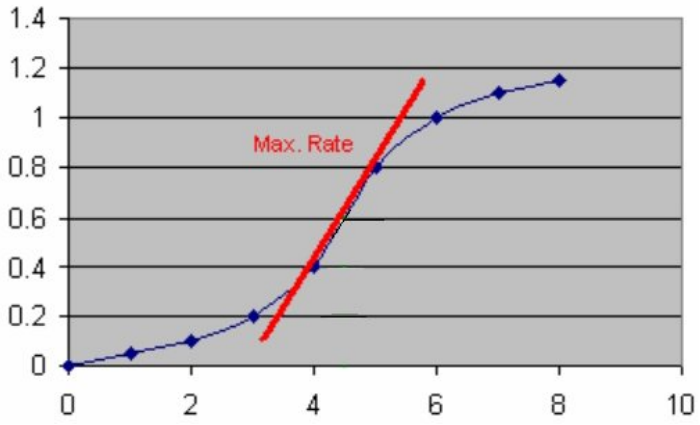
Kineettisiä laskentoja suoritetaan ainoastaan kineettisistä mittauksista. Ohjelmisto sisältää 11 erilaista kineettistä laskentaa. Tässä luvussa käsitellään ohjelmiston sisältämiä kineettisiä laskentoja.

6.4.1 Normaali nousunopeus

Normaalista nousunopeudesta voidaan käyttää myös termiä keskimääräinen nousunopeus (engl. average rate). Laskennassa selvitetään kaikkien mittaustulosten pohjalta, mikä on kineettisestä mittauksesta syntyvän käyrän keskimääräinen kulmakerroin. / 4, s. 132./

6.4.2 Suurin nousunopeus

Kun suurinta nousunopeutta selvitetään, selvitetään sitä hetkeä, jolloin kineettisestä mittauksesta muodostunut käyrä nousee nopeiten. Laskennassa sovitetaan käyrälle joukko suoria. Tarkastelun kohteena on näiden sovitettujen suorien suurin kulmakerroin. /4 s. 133./

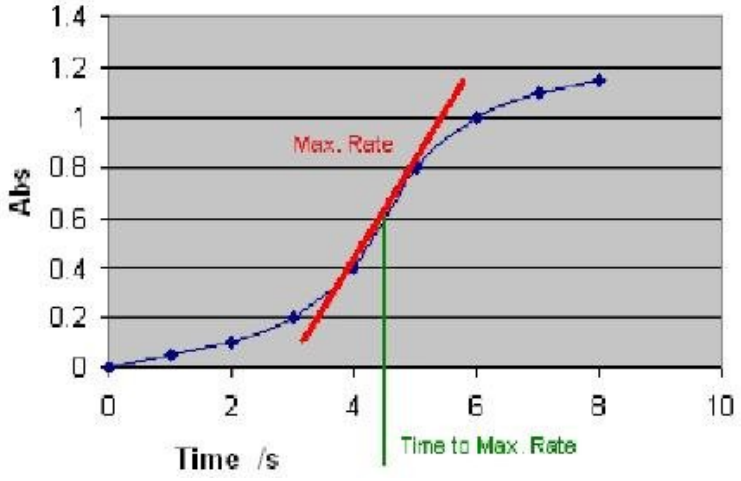


Kuva 5. Suurin nousunopeus /4, s. 133/

Kuvan 5 tilanne on esimerkki suurimmasta nousunopeudesta. Esimerkissä y-akselilla ovat mittaustulokset ja x-akselilla kulkee aika. Kuten edellä jo mainittiin, suurin nousunopeus on sama kuin käyrälle sovitettujen suorien suurin kulmakerroin. Kuvan 5 esimerkissä suurin nousunopeus on ajanhetkien 2,5-6 välillä.

6.4.3 *Suurimman nousunopeuden ajanhetki*

Suurimman nousunopeuden ajankohdan selvittäminen on hyvin samantapainen prosessi kuin suurimman nousunopeuden selvittäminen. Suurimman nousunopeuden ajanhetki on sen suoran puolivälissä, jolla on suurin kulmakerroin. /4, s. 134./



Kuva 6. Suurimman nousunopeuden ajanhetki /4, s. 134./

Kuvan 6 tilanne on esimerkki suurimmasta nousunopeudesta. Esimerkissä 6 on esitelty, missä kohtaa suoralta löytyy suurin nousunopeus. Kuten kuvasta 6 näkyy, ajan hetki, jolloin suurin nousu tapahtuu, on noin 5 sekunnin kohdalla.

6.4.4 *Muutosaika*

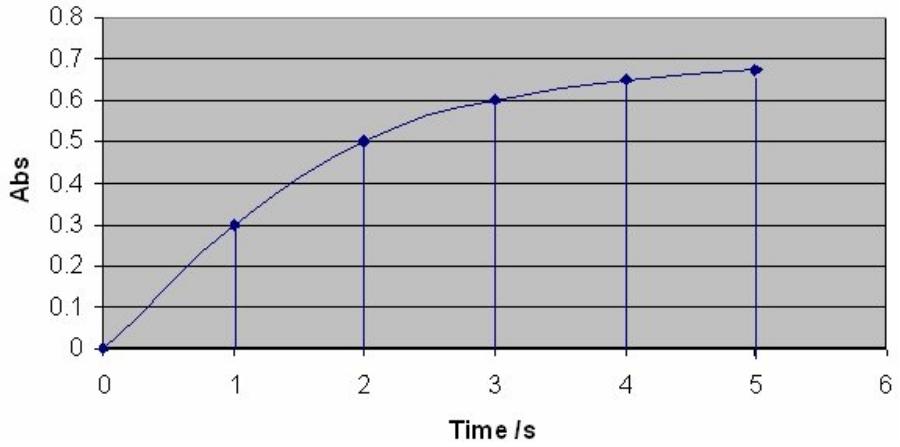
Kun selvitetään muutosaikaa, selvitetään, kuinka pitkä aika tarvitaan siihen, että ennalta määritetty muutos tapahtuu mitattavassa signaalissa. Tulos annetaan sekunneissa. /4, s.140./

6.4.5 *Suurin ja pienin arvo*

Suurinta arvoa voidaan etsiä kahdella eri tavalla. Voidaan etsiä jokaisen näytteen suurinta mitattua arvoa erikseen. Toinen vaihtoehto on etsiä koko mittaustapahtuman suurinta arvoa kaikista näytteistä. Kun määritellään pienintä arvoa, etsitään aina mittaustuloksen pientä arvoa. /4, s.139–140./

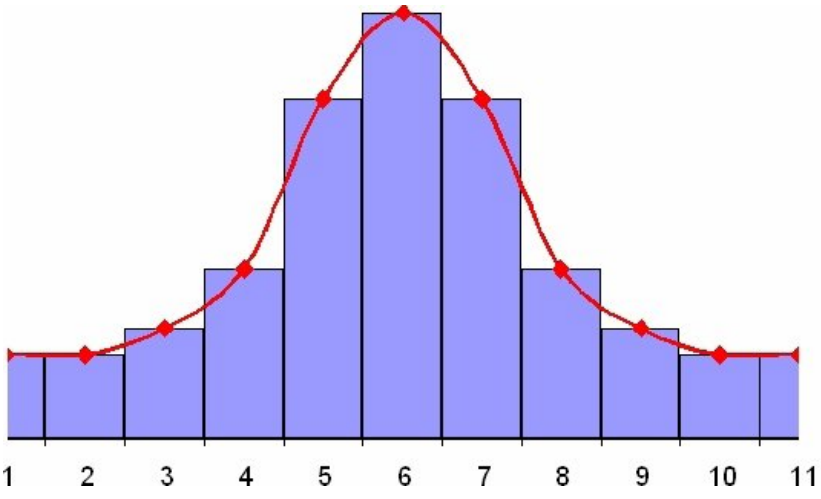
6.4.6 Summa, kineettinen keskiarvo ja integraali

Summa yksinkertaisesti laskee yhteen halutut lukemat. Otetaan esimerkiksi kuvan 7 tilanne. Kuvasta 7 on valittu tietyt pisteet yhteenlaskua varten. Valittujen lukemien summaksi tulee:
(0.00+0.30+0.50+0.60+0.65+0.67)abs=2.72abs



Kuva 7. Kineettinen yhteenlasku /4, s. 143./

Kun mittaustuloksista suoritetaan integraalilaskentaa, tarkoitetaan mittauspisteiden muodostaman alueen pinta-alaa (kuva 8).



Kuva 8. Kineettisen integraalin määritys /4, s. 142./

Kineettinen keskiarvo on kaikkien kineettisten mittaustulosten keskiarvo. Mittaustuloksia ei voi mitenkään rajata, vaan ohjelmisto laskee ihan kaikkien mittaustulosten keskiarvon.

6.5 Spektrin analysoiminen

Spektrin analysoimista käytetään spektrin skannauksesta syntyneen mittaustiedon käsittelyyn. Spektrin analysoimiseen liittyvissä laskennoissa oletetaan, että jokaista aallonpituutta kohden on vain yksi mittauspiste näytteessä. Ohjelmisto sisältää 7 erilaista laskentaa, jotka liittyvät spektrin analysoimiseen. Tämän luvun tarkoituksena on antaa käsitys, minkä tyyppisiä spektrin analysointiin liittyvät laskut ovat.

6.5.1 Spektrin piikin etsiminen

Ohjelmisto etsii mittauspiikkiä (engl. Spectral peak search) kaikista mitatuista aallonpituuksista jokaisessa näytteessä. Ohjelmassa voidaan määritellä raja-arvo. Kun mittaustulos ylittää tämän raja-arvon, voidaan se määritellä piikiksi. /5, s.177./

6.5.2 Spektrin suurin ja pienin arvo

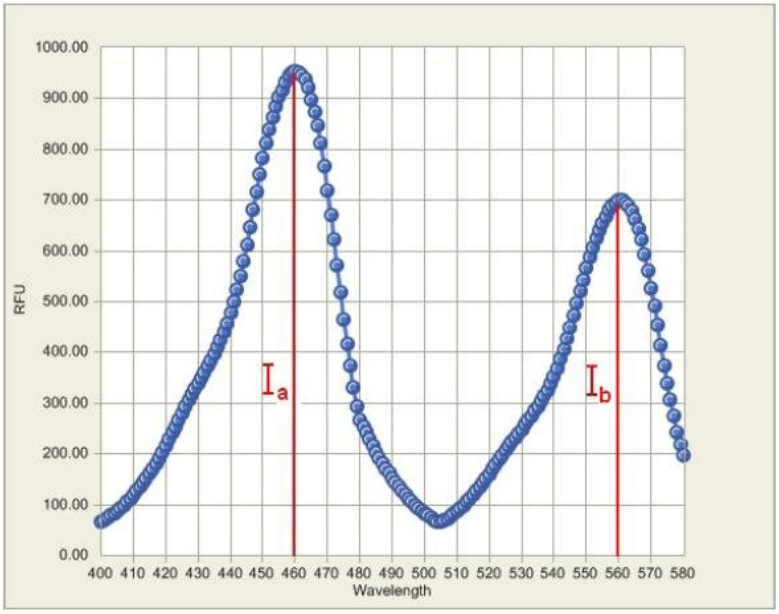
Spektrin suurin arvo tarkoittaa jokaisesta määritetystä näytteestä löytyvää suurinta mittaustulosta. Kun vastaavasti etsitään spektrin pienintä arvoa, etsitään sitä jokaisesta määritetystä näytteestä syntyneestä mittaustuloksesta. /5, s.179–180./

6.5.3 Suhdeluku spektrin sisällä

Kun selvitetään suhdelukua spektrin sisällä, selvitetään kahden aallonpituuden suhdelukua saman spektrin sisällä (engl. Ration within spectrum). Laskennat toteuttavat alla olevaa kaavaa, jossa I_a Ja I_b edustavat saman spektrin sisällä olevan kahden eri aallonpituuden intensiteettiä. /5, s.179./

$$X = \frac{I_a}{I_b} \tag{5}$$

Esimerkiksi kuvan 9 esimerkissä aallonpituudet ovat n_a ja n_b, joiden intensiteetit on annettu kuvassa I_a Ja I_b.



Kuva 9. Suhdeluku spektrin sisällä /5 s. 179/

Jos kuvan 9 esimerkkitilanteessa lasketaan suhdeluku, näiden kahden eri aallonpituuden välillä se olisi: 900 /700 =1,28.

6.6 Puoliintumisaikalaskennat

Puoliintumisaikalaskennoissa (engl. Decay Calculation) keskitytään selvittämään Tau ja Alpha arvot mittaustuloksista. Alpha edustaa mitatun signaalin intensiteetin huippuarvoa ajan hetkellä 0. Tau edustaa sitä signaalin arvoa, jolloin sen intensiteetti on laskenut puoleen huippuarvosta. Puoliintumisaikalaskennat pohjautuvat aina aikaerotteiseen fluorometriseen hajontamittaukseen (engl. Time Resolved Fluorometric Decay measurement). /5, s. 203./

7 TESTIYMPÄRISTÖ

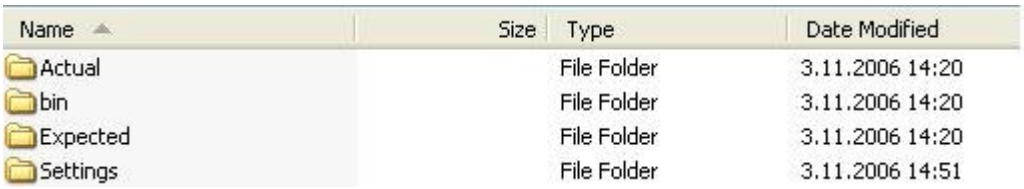
Testauksen automatisoinnissa käytettiin TestReflector-nimistä ohjelmaa. TestReflector on yrityksen kehittämä moduulitestaukseen tarkoitettu työkalu. Luvussa käydään läpi kuinka TestReflector toimii.

7.1 TestReflector

Kun TestReflectorin kehitystyö aloitettiin yrityksessä, pääideana oli kehittää ohjelma, joka pystyisi testaamaan moduulin toimintaa ilman testiajuria tai tynkämoduuleja. Valitettavasti tämän hetkinen TestReflectorin versio ei tähän vielä pysty kaikissa tapauksissa. TestReflectorissa ei ole graafista käyttöliittymää, vaan se toimii komentokehotetilassa. TestReflector.exe käskyn syntaksi on seuraavanlainen:

TestReflector.exe Testattava.dll testiasetukset.xml

TestReflector on hyvin tarkka siitä, missä hakemistossa testauksessa käytettävät tiedostot sijaitsevat. Kuva 10 kertoo TestReflectorin käyttämän hakemistorakenteen. /10, s. 2/



Kuva 10. TestReflectorin hakemistorakenne

Testauksesta syntyvät testitulokset generoituvat Actual-hakemistoon. Bin-hakemistossa sijaitsee varsinainen TestReflector.exe ja samassa hakemistossa pitää olla myös testattava dll-tiedosto. Expected-hakemisto sisältää testien oikeat tulokset. TestReflectorin käyttämät testausohjeet sijaitsevat Settings-hakemistossa, jotka ovat XML-muodossa. /10, s.3./

TestReflector on kirjoitettu C#-kielellä. TestReflectorin toiminta pohjautuu C#-kielessä käytettyihin heijastumiin (engl. reflection). Heijastumien avulla TestReflector saa testattavan luokan metodit ja muuttujat käyttöönsä riippumatta siintä, mikä niiden näkyvyysmääre on. Heijastumien avulla on esimerkiksi mahdollista saada käyttöön sellaisen luokan metodit, jonka näkyvyysmääre on internal tai private. TestReflector saa tiedon, mitä

luokkaa ja mitä luokan metodeita testataan Settings-tiedostosta. Settings-tiedoston toimintaa käsitellään tarkemmin kappaleessa 7.2. /10, s. 3./

7.2 Settings-tiedosto

Settings-tiedosto on XML-tiedosto, joka sisältää tiedon mitä luokkaa ja mitä sen metodeja tullaan testaamaan. XML-tiedostossa kerrotaan ne parametrit, jotka välitetään testattavalle metodille. Kuvassa 11 esitellään yksikertainen settings-tiedosto, joka testaa TestClass nimisen luokan Mul-metodin toimintaa:

```
<?xml version="1.0" encoding="utf-8"?>
<data>
    <class name="TestClass">
        <method name="Mul">
            <System.Int32>1</System.Int32>
            <System.Int32>2</System.Int32>
        </method>
        <method name="Mul">
            <System.Int32>-3</System.Int32>
            <System.Int32>4</System.Int32>
        </method>
    </class>
</data>
```

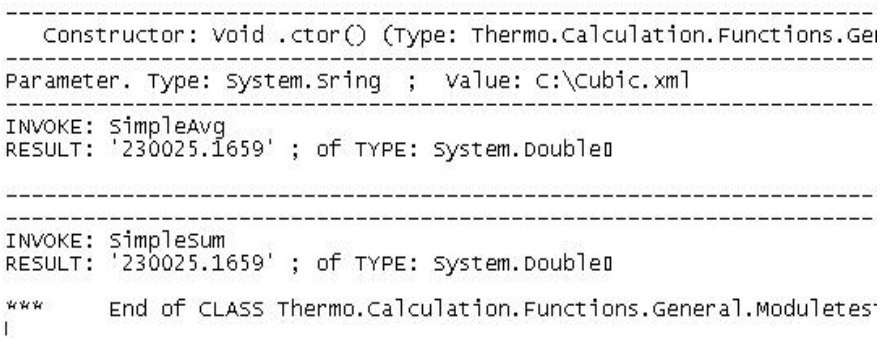
Kuva 11. Settings-tiedosto

Class-tunnisteiden name-attribuutin avulla kerrotaan testattavan luokan nimi. Method-tunnisteiden ja name-attribuutin avulla kerrotaan mitä luokassa olevia funktioita halutaan testata. Method aloitus- ja lopputunnisteiden sisällä kerrotaan, minkä tyyppisiä parametrejä halutaan välittää testattavalle funktiolle. Esimerkiksi kuvassa 11 välitetään Mul-metodille kaksi int-tyyppistä lukua. Settings-tiedoston avulla pystytään välittämään minkä tahansa C#-kielessä käytössä olevan muuttujan tietotyyppi. Jos esimerkiksi vaihtoehtoisesti halutaan välittää Double–tyyppisiä lukuja, ilmaistaan se tiedostossa seuraavasti:

<System.Double>343.56</System.Double>

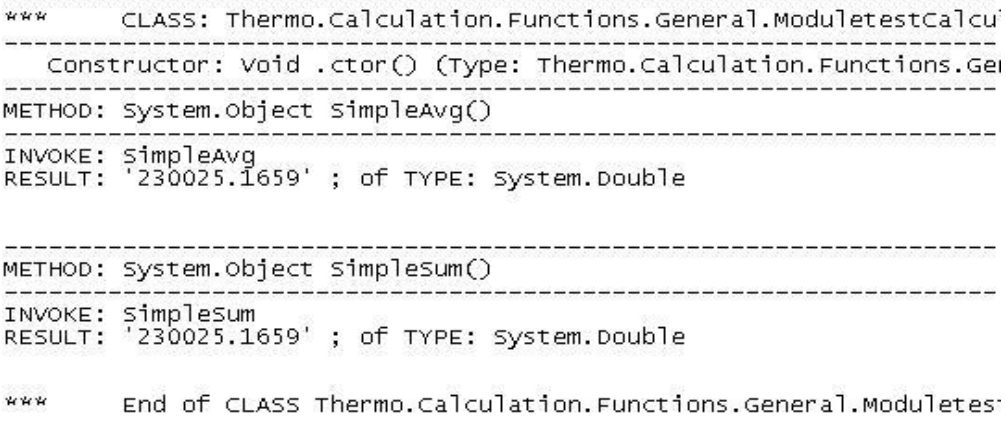
7.3 Actual- ja Expected-lokitiedostot

Actual-lokitiedosto (kuva 12) generoituu aina testin suorituksen yhteydessä hakemistoon ***\TestReflector\Actual***. Actual-lokitiedosto sisältää tiedot muuttujien arvoista, joita TestReflector on vienyt testattavalle luokalle ja tiedon testattavan funktion palauttamasta testituloksesta. /10, s. 5./



Kuva 12. Actual-lokitiedosto

Toinen TestReflectorin tarvitsema lokitiedosto on nimeltään Expected (kuva 13). Expected-lokitiedosto ei generoidu automaattisesti kuten Actual-lokitiedosto. Se pitää tehdä itse käsin, ennen kuin testejä voidaan alkaa suorittaa. Expected-lokitiedosto sisältää tiedon mitä testitulokseksi pitäisi tulla. Expected-lokitiedostojen pitää sijaita hakemistossa ***\TestReflector\Expected***, jotta TestReflector osaa etsiä sitä oikeasta osoitteesta. /10, s. 5./



Kuva 13. Expected-lokitiedosto

TestReflector tekee päätöksen onko testi onnistunut vai ei näiden kahden lokitiedoston avulla. TestReflector etsii näiden kahden tiedoston väliltä eroavaisuuksia. TestReflector käy molemmat lokitiedostot läpi ja etsii näiltä riveiltä tiettyjä avainsanoja. TestReflector vertailee, ovatko rivit, joilla avainsanat sijaitsevat samat ja ovatko ne samalla rivillä kummassakin lokitiedostossa. Esimerkiksi jos Expected-lokitiedossa Result-avainsana löytyy riviltä 12, pitää kyseisen Result-avainsanan olla myös Actual-lokitiedossa rivillä 12.

Kahteen lokitiedostoon perustuvan päätöksentekomenetelmä on nykyisen Testreflectorin version ehdoton Akilleen kantapää kahdesta syystä. Ensinnäkin Expected-lokitiedosto pitää luoda käsin teksti-editorilla. Lokitiedoston luonti käsin on todella hankalaa ja aikaa vievää työtä. Toiseksi

tiettyjen rivien pitää olla kummassakin lokitiedostossa oikeassa kohdassa, jotta lokitiedostojen tarkastelu saadaan automaattisesti läpi. Tämä aiheuttaa paljon testien turhia epäonnistumisia, vaikka testattava metodi olisikin suorittanut tehtävänsä aivan oikein. Syynä tähän voi olla pelkästään käsikirjoitettuun Expected-lokitiedostoon vahingossa tullut yksi ylimääräinen rivi.

8 TESTIEN SUUNNITTELU JA TOTEUTUS

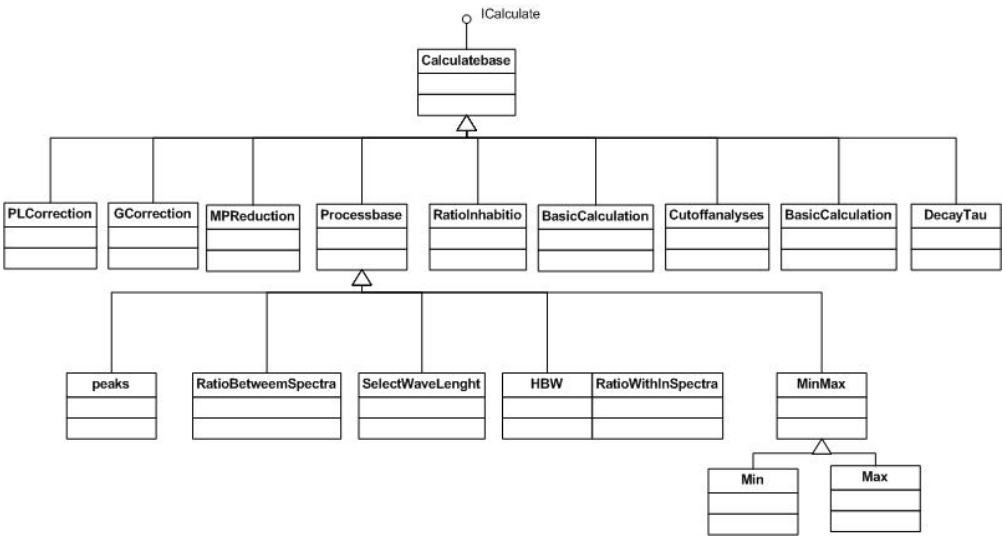
Luvussa tärkein asia on laskentojen testien suunnittelun ja toteutuksen kuvaaminen. Näiden asioiden lisäksi luvussa käsitellään, kuinka ohjelmisto suorittaa laskennat ja minkälaisista luokista laskennat koostuvat ja mikä on luokkien välinen suhde.

8.1 Yleistä ohjelmiston laskennoista

Tässä kappaleessa käsitellään ohjelmiston laskentoja toteuttavien luokkien välisiä suhteita. Tarkastellaan tapahtumaa sekvenssikaavion avulla, miten ohjelmiston laskentojen suorittaminen etenee. Luvun lopussa esitellään tapa, jolla testauksessa käytetty testiaineisto generoidaan ja miten saadaan aikaiseksi oikeat tulokset (engl. expected result).

8.1.1 Ohjelmiston laskentojen luokkakaavio

Testitapausten suunnitteluissa nojauduttiin kahteen seikkaan. Ensinäkin käytettiin hyväksi yrityksen manuaalitestausjärjestelmää. Toisena asiana testien suunnittelussa mukauduttiin vahvasti myös ohjelmiston toteutukseen.



Kuva 14. Laskentojen luokkakaavio

Kuvassa 14 on luokkakaavio ohjelman laskennoista. Kuten kuvasta näkyy, kaikkiin laskentoihin pääsee käsiksi ICalculate-rajapinnan kautta. Rajapintaa toteuttaa Calculate-funktion. Calculate-funktion avulla pystytään

toteuttamaan kaikki ohjelmistossa tapahtuvat laskennat. Calculate-funktion prototyyppi on

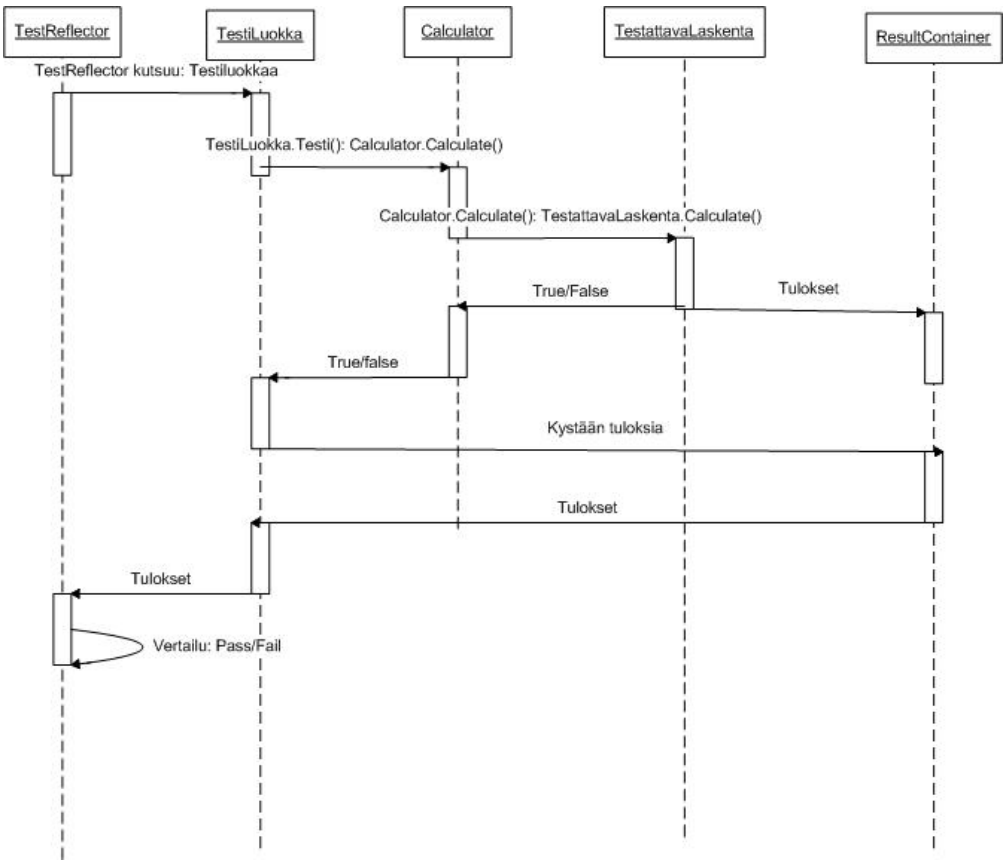
bool Calculate(ResultContainer result, string xmlnode)

Ensimmäisenä parametrina Calculate-funktio haluaa ilmentymän ResultContainer-luokasta. ResultContainer-luokan sisältää mittaustulokset ja laskennoista syntyneet tulokset. ResultContainer-luokasta löytyy kappaleesta 8.2.1.

Toisena parametrina Calculate-funktiolle pitää välittää xml-tunniste merkkijonona. Tämä tunniste sisältää tiedon siintä, minkä tyyppinen laskenta on kyseessä. Työssä tästä parametrista käytettiin nimeä Testiformula. Tarkempi kuvaus Testiformulasta löytyy kappaleesta 8.2.3.

8.1.2 Laskentojen testauksen kulku

Hyvän ICalculate-rajapinta toteutuksen vuoksi eri laskentaoperaatioiden testaaminen oli hyvin suoraviivaista työtä. Lähes kaikki laskennat perivät ICalculate-rajapinnassa saman Calculate-funktion. Tyypillistä laskentojen testauksen kulkua kuvaa kuva 15.



Kuva 15. Tapahtumasekvenssikaavio testauksen kulusta

Kuvassa 15 TestReflector ottaa käyttöönsä testauksessa käytetyn testiohjelman funktiot ja alkaa suorittaa niitä. Testiohjelmassa luodaan aina instanssi Calculator-luokasta, jossa ICalculate-rajapinta on toteutettu. Testiohjelman avulla välitetään ICalculate-rajapinnan kautta testin kohteena olevalle laskennalle kaikki sen tarvitsemat mittaustulokset ja parametrit. Testattava laskenta suorittaa annettujen tietojen avulla laskennat, ja palauttaa paluuarvona takaisin Calculator-luokalle tosi (true), jos laskenta on onnistunut, tai epätosi (false), jos jotakin on mennyt vikaan. Laskentojen tulokset löytyvät ResultContainer-luokan ilmentymästä.

8.1.3 Testiarvot ja odotettavissa olevat tulokset

Testauksessa käytettävät testiarvot saadaan suoraan mittalaitteiden mittaustuloksista. Nämä mittaustulokset haetaan ohjelman käyttämästä Microsoft SQL-palvelimelta XML-tiedostona.

Oikeat referenssitulokset saadaan, kun suoritetaan laskennat sellaisella ohjelmaversiolla, jolla laskenta on testattu ja todettu toimivaksi.

8.2 Testauksen valmistelu

Tämän kappaleen tarkoituksena on käsitellä tiettyjä asioita, jota toistuvat jatkuvasti eri laskentojen testaamisessa. Kappaleessa käydään läpi Setup-funktion rakenne ja sen toiminta. Kappaleessa viimeisenä aiheena on TestFormula-muuttuja ja sen merkitys ohjelmiston testauksen kannalta.

8.2.1 ResultContainer-luokka ja ResultItem-luokka

Lähes kaikkien laskentojen suorittamiseen ohjelmassa tarvitaan yksi ilmentymä ResultContainer-luokasta ja kaksi ilmentymää ResultItem-luokasta. yhtä ResulItem-luokan ilmentymiä tarvitaan mittaustulosten ja mittausparametrien tallentamiseen ja toista laskennoista syntyneiden tulosten varastoimiseen. ResultContainer-luokan ilmentymä on koosteolio, joka koostuu kummastakin ResultItem-luokan ilmentymästä.

Laskentojen testaaminen ei tee poikkeusta tästä. Lähestulkoon kaikissa laskentoihin liittyvissä testeissä käytetään tätä rakennetta. Ainoan poikkeuksen tekee Simplecalculation-projekti, jonka testaamiseen ei tarvita tätä rakennetta. SimpleCalculation-projektin testaamisen suunnittelu ja toteutus on kuvattu kokonaisuudessaan luvussa 8.3.

Kun jotakin laskennan testaamista varten luotiin ilmentymä ResultContainer-luokasta, käytetään nimeä TestContainer. TestContainer oli koosteolio ja koostuu kahdesta ResulItem-luokan ilmentymästä. Näistä ilmentymistä käytettiin nimiä Input ja Output. Input-instanssiin varastoitui kaikki mittaustulokset ja muut mittausparametrit. Output-Instanssiin varastoituivat kaikki laskentojen tulokset.

8.2.2 Setup-funktion toteutus

Ohjelman käyttämästä tietokannasta saa tuotua ulos jonkin mittauksen tulokset, ja kyseiseen mittaukseen liittyvät muut parametrit. Setup-funktion tehtävänä on kerätä kaikki ne tiedot, jotka vaaditaan jonkin tietyn laskennan suorittamiseen. Kaikissa testipedeissä Setup-funktion prototyyppi esiintyy muodossa

- public void Setup(string polku);

Setup-funktio saa parametrikseen merkkijonon. Tämä parametrin avulla välitetään tieto, missä testauksessa käytettävä XML-dokumentti sijaitsee.

XML-tieto ladataan aina .NET-sovelluskehyksen tarjoaman XMLDocument-luokan instanssiin. Kun kaikki mittausarvot ja parametrit on ladattu XMLDocument-luokan instanssiin, voidaan valita sieltä tarpeelliset tiedot. Valinta tapahtuu luomalla instanssi XMLNodeList-luokasta. Tähän instanssiin voidaan tallentaa haluttu tunniste kaikkine mahdollisine alitunnisteineen. Voisi sanoa, että XMLNodeList-luokan avulla muodostetaan oma tilapäinen pikku XML-dokumentti jatkokäsittelyä varten. Otetaan esimerkkinä kuvan 16 tilanne. Kun XMLNodelist-luokan instanssin avulla halutaan ladata tbl_result-tunniste, latautuvat siihen kaikki alitunnisteet väliltä *<tbl_results> </tbl_results>.*

```
- <tbl_results>
    <id>19300</id>
    <executed_step_id>892d1a33-59db-4540-90f2-a62bceaac296</executed_step_id>
    <MeasureName>Pho-Scanning1</MeasureName>
    <replicate_id>a489bff2-e6df-4d5c-84d2-0220e59663d7</replicate_id>
    <iteration>1</iteration>
    <ui_format>0</ui_format>
    <ui_saturation>0</ui_saturation>
    <ul_ref_time>847398</ul_ref_time>
    <originalValue>0.00500504</originalValue>
    <well_x>0</well_x>
    <well_y>0</well_y>
    <point_x>0</point_x>
    <point_y>0</point_y>
    <exitWL>400</exitWL>
    <emissionWL>0</emissionWL>
    <disabled>0</disabled>
    <polarizationType xml:space="preserve"></polarizationType>
    <plate_name>1</plate_name>
  </tbl_results>
```

Kuva 16. Esimerkki XmlNodeListin käytöstä

Tämän jälkeen XmlNodelist-luokan instanssista poimitaan kaikkien alitunnisteiden arvot foreach-silmukan avulla ja asettaa ne omalle paikalleen ResulItem luokan instanssiin Input.

8.2.3 TestFormula-muuttuja

Testformula-muuttuja on ohjelmiston laskentojen testaamisen kannalta hyvin tärkeä. Sen tärkein tehtävä on kertoa ICalculate-rajapintaa toteuttavalle Calculate-funktiolle, mikä ohjelmiston laskenta on kyseessä.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
- <UIResultRelations>
    <PlateIds />
  - <Result id="cd538f1a-5740-465b-8345-9dcab486fdd9" name="TRF-Decay1" type="TRFDecay">
      <ControlLayout />
    - <Result id="37c511a3-a639-454c-ba0f-1dcc963b6ddc" name="DecayCalc1" type="DecayCalc">
        <ControlLayout />
      - <Calculation funcAssemblyName="Calculation" func="Thermo.Calculation.Functions.TRFDecay.DecayTau">
          <CalculationType>DecayCalc</CalculationType>
          <ValuesTableName>Input</ValuesTableName>
          <ResultTableName>Output</ResultTableName>
        </Calculation>
      </Result>
    </Result>
  </UIResultRelations>
```

Kuva 17. TestFormula-muuttuja

TestFormula-muuttuja on merkkijono. Kuten kuvasta 17 voi päätellä, TestFormula-muuttuja on käytännössä XML-dokumentti. TestFormula-muuttuja välitetään Calculate-funktiolle merkkijonon muodossa.

XML-tunniste ValuesTableName kertoo tiedon mikä nimisestä ResultItem-luokan ilmentymästä löytyvät mittaustulokset. XML-tunniste ResultTableName kertoo minkä niminen ResulItem-luokan instanssi pitää sisältää laskennoista syntyneet tulokset. Calculate-tunnisteen toisen parametrin avulla kerrotaan laskennan tyyppi.

8.3 Simplecalculation-laskennat

Simplecalculation-projekti koostuu viidestä luokasta. Avg-luokka laskee keskiarvon jollekin lukujoukolle. Sum-Luokka laskee lukujoukon yhteenlaskettujen arvojen summan. Min-luokka etsii lukujoukon pienimmän arvon. Max-luokka etsii lukujoukon suurimman arvon. MaxMinusMin-luokan tehtävänä vähentää pienimmän arvon suurimmasta.

8.3.1 Simplecalculation-testien suunnittelu

Mitään virallista olemassa olevaa testaussuunnitelmaa ei ole olemassa, joten siihen ei voitu nojautua. Tässä kohtaa nojauduttiin testiä suunniteltaessa ainoastaan tekniseen toteutukseen.

Näitä viittä luokkaa toteuttaa ISimpleCalculation-rajapinta. ISimpleCalculation-rajapinnasta nämä viisi luokkaa perivät Calculate-funktion, jota käytetään laskutoimitusten suorittamiseen. Calculate-metodi esiintyy rajapinnassa muodossa:

```
object Calculate(DataRow [] rowsValues, string sColumnName,
ref Type typeResult);
```

Calculate-luokka vaatii kahdeksi ensimmäiseksi parametriksi *tietueen (engl. Datarow)* ja sarakkeen nimen*.* Nämä kaksi parametria ovat kumpikin C#-kielessä käytetyn taulukkotietotyypin (engl. Datatable) osia. Se koostuu riveistä ja sarakkeista. Käytännössä Datatable-rakenne on kuin mikä tahansa taulukkorakenne. Calculate-metodille on tarkoitus välittää ainoastaan mittaustulokset. Viimeisenä parametrina pitää Calculate-metodille välittää parametri *ref typeResult*. Määreiden *ref Type* avulla välitetään osoitin, joka kertoo minkä tyyppisiä arvoja on tulossa.

8.3.2 Simplecalculation-projektin testien toteutus

Ensimmäinen toimenpide oli luoda testiohjelma, jota käytetään ainoastaan Simplecalculation-projektin luokkien testaamiseen. Jokaisen Simplecalculation-projektin testaamista varten oli tehty testaamiseen tarvittava funktio.
Testifunktiot loivat ensimmäisenä toimenpiteenä instanssin testin kohteena olevasta luokasta heijastumien (engl. Reflection) avulla. Luokkien instanssit pitää luoda heijastumien avulla, koska kaikkien projektin luokkien näkyvyysmääre on Internal. Ilman heijastumien käyttöä luokkien testaaminen ei ole mahdollista.

Toisena tärkeänä toimenpiteenä funktiot keräävät testaamisessa käytettävän testitiedon XML-dokumentista. Testitiedosta ne muodostavat testaamisessa tarvittavan taulukon, joka sisältää yhden sarakkeen ja n kappaletta rivejä.

Testien toteutuksen viimeisenä osana oli liittää testattavat funktiot osaksi settings-tiedostoa. Kuva 18

```
- <class name="SimpleCalculationTest">
    <method name="SimpleAvg" />
    <method name="SimpleSum" />
    <method name="SimpleMinValue" />
    <method name="SimpleMaxValue" />
    <method name="SimpleMaxMinusMin" />
  </class>
```

Kuva 18. SimpleCalculationTest testiohjelman lisääminen osaksi Settings-tiedostoa

Kuvassa 18 on class-tunnisteen parametrina testiohjelman nimi. Method-tunnisteen avulla välitetään tieto kaikista testauksessa mukana olevista testimetodeista.

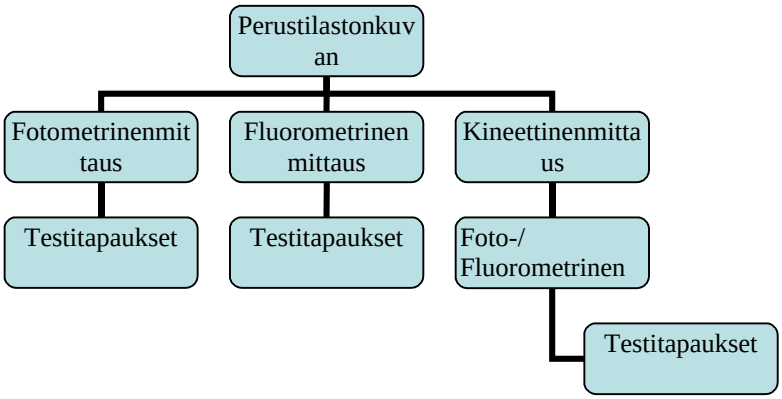
8.4 Perustilastotietotestit

Perustilastotietolaskennoissa keskitytään selvittämään keskiarvoa, yleistä poikkeamaa ja vaihtelukerrointa mittaustuloksista. Tässä osuudessa käsiteltiin perustilastotietotestien toteutusta ja suunnittelua. Laskentojen kuvaus löytyy kappaleesta 6.1.

8.4.1 Perustilastotietotestien suunnittelu

Perustilastontietolaskentoja toteuttaa BasicStatic-luokka. Luokassa on toteutettu yksi funktio, joka hoitaa kaikkien näiden kolmen arvon laskemisen ja tulosten tallentamisen ResultContainer-luokan ilmentymään. Kuten kappaleessa 8.1, ainoa keino päästä suorittamaan näitä laskentoja oli ICalculate-rajapinnan kautta.

BasicStatic-luokan laskentojen pitää toimia kaikilla mittaustekniikoilla. Calculate-funktion pitää osata laskea oikeat tulokset kaikissa mittausteknologia vaihtoehdoissa. Kineettinen mittaus ei myöskään vaikuta millään tavalla laskujen suorittamiseen. Kuvassa 19 on esitelty, miten BasicStatic-luokan testitapaukset jakautuvat.



Kuva 19. Perustilastotiedon testitapauskaavio

Kuvassa 19 esitetään, miten testitapaukset jakautuvat kolmen mittausteknologian mukaan. Testit testaavat jokaisella mittausteknologialla, laskeeko ohjelma kaikki perustilastotietolaskut oikein. Testaus suoritetaan myös kineettisesti kummallakin mittausteknologialla.

8.4.2 Perustilastotietotestien toteutus

Testauksessa käytettävän tiedon käsittelyä varten tarvittiin Setup-funktio, jonka toteutus on kuvattu luvussa 8.2.3. Jokaista suureen testaamista varten tarvittiin jokaiselle omat funktionsa:

- `public double GetCV() CV:n testaamista varten,`
- `public double GetSD() SD:n testaamista varten ja`
- `public double GetAvg() keskiarvon testaamista varten`

Toteutukseltaan nämä funktiot olivat äärimmäisen yksinkertaisia, niissä kutsuttiin Calculate-luokan ilmentymän avulla ICalculate-rajapinnassa olevaa Calculate-funktiota. Calculate-funktion avulla välitettiin mittaustulokset laskentaa toteuttavalle luokalle. Luokka suoritti laskennat annettujen tietojen pohjalta ja palautti tuloksen takaisin testifunktiolle. Testifunktio palautti tuloksen TestReflectorin käyttöön, joka teki päätöksen, onko testi onnistunut vai ei.

Viimeisenä vaiheena oli perustilastotietotestien liittäminen osaksi Settings-tiedostoa. Kuva 20 osoittaa kuinka testit näkyvät Settings-tiedostossa.

Kuva 20. Perustilasto testien liittäminen osaksi automatisointia

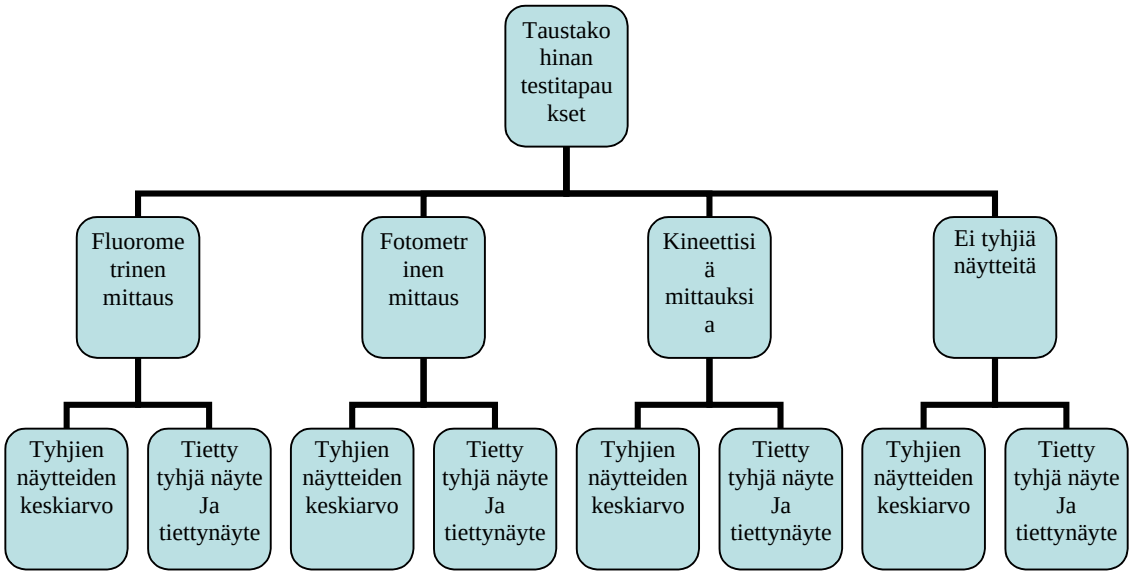
Kuva 20 esittää myös, kuinka perustilastotietotestit näkyvät osana Settings-tiedostoa. Kuten kuvasta 20 näkyy, Setup-funktiota kutsutaan kolme kertaa. Jokaisella kutsukerralla välitetään eri mittausteknologioilla tehdyt mittaustulokset. Tämä täytyy tehdä, jotta kuvan 19 testitapauksien hierarkiakaavion kaikki haarat saadaan toteutumaan.

8.5 Taustakohinan vähentämisen testaaminen

Tässä Kappaleessa käsitellään taustakohinan vähentämiseen (engl. blank subtraction) liittyvien testien suunnittelua ja toteutusta. Taustakohinan vähentämistä laskutoimituksena käsiteltiin kappaleessa 6.2.

8.5.1 Taustakohinan vähentämisen testauksen suunnittelu

Taustakohinan vähentämisen testaamisessa perusperiaate on se, että sen pitää toimia kaikilla mittaustekniikoilla ja kaikilla kineettisillä mittauksilla. Mittaustekniikka ei siis saa olla este tulosten syntymiselle. Oikeastaan ainoa asia, mikä estää mittaustulosten syntymisen on se, että mittausta tehdessä ei ole yhtään tyhjää näytettä (engl. blank).



Kuva 21. Taustakohinan testitapausten hierarkia

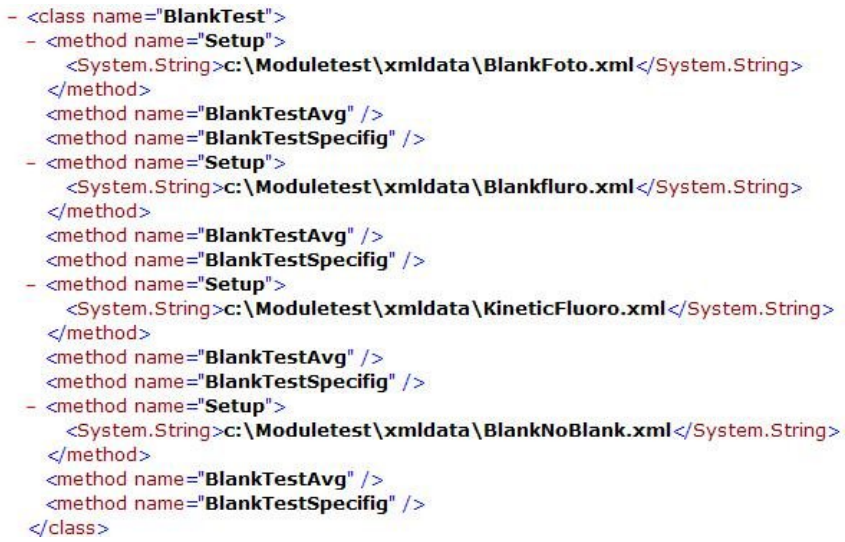
Kuvassa 21 on esitelty taustakohinan testauksen testitapaushierarkia. Kuvan ideana on se, että kaikilla mittausteknologioilla suoritettujen mittausten pohjalta pitää syntyä tuloksia, jos mittauksissa on tyhjiä näytteitä. Jos mittauksessa ei ole yhtään tyhjää näytettä, tuloksia ei silloin myöskään synny. Kuvassa 21 tätä tilannetta vastaa oikeanpuoleisin haara.

8.5.2 Taustakohinan vähentämisen testauksen toteutus

Laskentojen testaamista varten pitää tuottaa neljät eri mittaustulokset kaikille taustakohinan mittausteknologia tyypille, jossa on tyhjiä näytteitä. Lisäksi yksi sellainen mittaus, jossa ei ole tyhjiä näytteitä. Mittauksissa on määritetty joukko tyhjiä näytteitä, joista lasketaan keskiarvo. Se vähennetään varsinaisesta mittaustuloksesta ja määritetään tyhjät näytteet vastaamaan jotakin tiettyä näytettä. Ohjelman suorituslogiikka osaa tehdä itse päätöksen, milloin se haluaa vähentää tyhjien näytteiden keskiarvon kaikista mittaustuloksista tai vähennetäänkö jonkin tietyn tyhjän näytteen arvo jostakin tietystä mittaustuloksesta. Ainoa asia mikä testauksen aikana pitää tehdä, on vaihtaa parametrina välitettävää XML-tiedostoa.

Laskentatyyppien testaamiseen tarvittiin kaksi testifunktiota. Funktio BlankAvg testaa tyhjien näytteiden vähentämistä mittaustuloksesta. Toinen funktio on BlankSpecifig, joka testaa tietyn tyhjän näytteen vähentämistä tietystä mittaustuloksesta.

Lopuksi nämä suunnitellut testit liitetään osaksi automatisoitavaa kokonaisuutta.



Kuva 22. Taustakohinatestien lisääminen osaksi automatisointia

Kuvassa 22 laskennan testitapaukset näkyvät osana Settings-tiedostoa. Kuten kuvasta näkyy, Setup-funktiota kutsutaan neljä kertaa. Kaikilla Setup-funktion kutsukerroilla sille välitetään parametrina eri mittausteknologialla suoritettuja mittauksia. Näin saadaan käytyä kaikki kuvan 21 haarat läpi

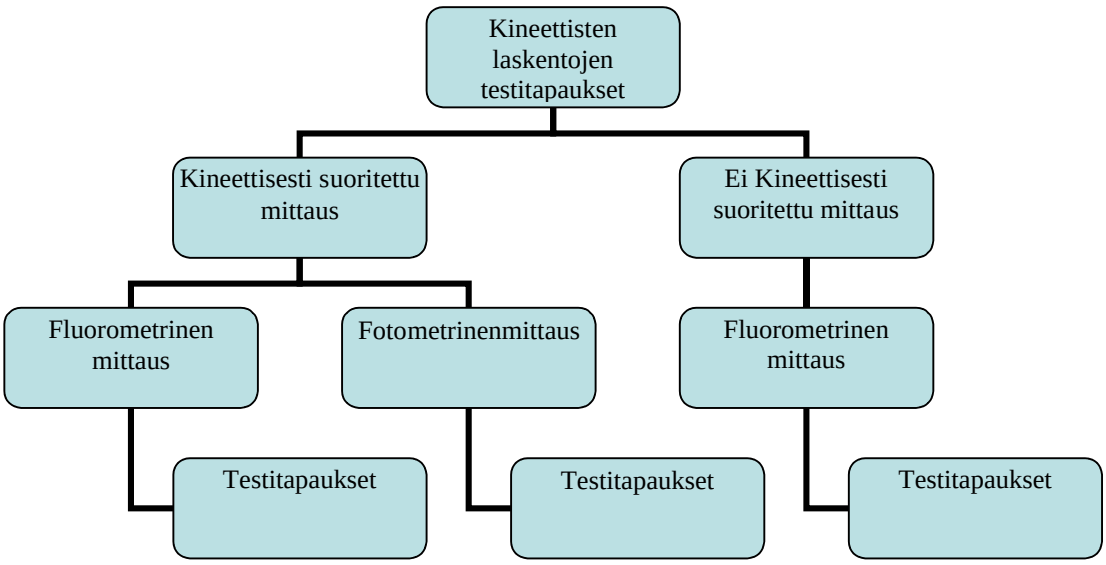
8.6 Kineettisten laskentojen testaaminen

Tässä luvussa käsitellään kineettisten laskentojen suunnittelua ja testaamista. Kineettiset laskennat esiteltiin kappaleessa 6.4.

8.6.1 Kineettistenlaskentojen testaamisen suunnittelu

Kuten kappaleessa 6.4 todettiin, kineettiset laskennat koostuvat kahdeksasta erilaisesta laskutoimituksesta. Jokaisen näiden laskutoimituksen testaamiseen löytyy yrityksestä manuaaliset testaussuunnitelmat ja niiden tekninen toteutus.

Luvussa 6.4 todettiin, että kineettiset laskennat on tarkoitettu käsittelemään kineettisiä mittauksia. Kineettisten laskentojen suorittaminen ohjelmassa ei ole mahdollista, ellei ole kyseessä jokin kineettinen mittaus. Mittausteknologia ei ole este laskentojen suorittamiseen, kunhan se vain on suoritettu kineettisesti.



Kuva 23. Kineettisten laskentojen testitapausten hierarkia

Edellä esiteltyjen määrittelyjen pohjalta on johdettu kuvan 23 testitapaukset. Kuvassa 23 kaikki kineettisesti suoritettujen mittausten pohjalta tehdyt laskennat antavat tuloksen myös testauksessa (kuvan 23 vasen puoli). Ei-kineettisesti suoritetut mittaukset eivät saa antaa minkäänlaisia laskentatuloksia (kuvan 23 oikea puoli). Laskentoja suorittavan logiikan pitäisi katkaista laskentojen suorittaminen ennen kuin mitään laskentaa edes aletaan suorittaa. Kuvassa esitelty testitapaukset-laatikko pitää sisällään kaikkien kineettisten laskentojen testitapaukset.

8.6.2 Kineettisten laskentojen testaamisen toteutus

Kineettisten testien toteuttaminen aloitettiin luomalla oma testiluokka. Seuraavaksi tarvittiin mittaustiedon käsittelyä ja tietorakenteiden alustamista varten oma Setup-funktio. Setup-funktion toteutus esiteltiin kappaleessa 8.2.3.

Jokaista laskentaa varten luotiin omat testifunktiot. Toteutukseltaan nämä funktiot ovat hyvin samanlaisia, mutta niissä on yksi oma erikoispiirteensä. Jokaisessa kineettistä laskentaa testaavassa funktiossa pitää olla oma testiformula-muuttuja, jonka näkyvyys on vain oman funktionsa sisällä. Kappaleessa 8.2.3 käsiteltiin TestFormula-muuttujaa.

TestFormula-muuttuja toimii ihan samalla tavalla kuin muidenkin laskentojen testaamisessa, mutta kineettisten laskentojen kohdalla TestFormula-muuttuja välittää tiedon siitä, että on kyseessä kineettinen laskenta ja myös tiedon, minkä tyyppinen kineettinen laskenta on kyseessä.

```
1  <Calculation>
2      <CalculationType>KineticProcessor</CalculationType>
3      <ValuesTableName>f123dbd8-a8e7-43b1-917b-8f69991c8e66</ValuesTableName>
4      <ProcessorType>TimeToMaxRate</ProcessorType>
5      <Reaction>Undefined</Reaction>
6      <KineticRate>s</KineticRate>
7      <IgnoreFirstIIReadings>0</IgnoreFirstIIReadings>
8      <IgnoreLastIIReadings>0</IgnoreLastIIReadings>
9      <Window>2</Window>
10     <Baseline>0</Baseline>
11     <Change type="Absolute">0</Change>
12     <ReadingIndex>1</ReadingIndex>
13 </Calculation>
```

Kuva 24. Kineettisen laskennan parametrit

Kuvassa 24 rivillä 4 on tieto minkä tyyppinen kineettinen laskenta on kyseessä, ja rivin 4 alla minkä muun tyyppisiä parametrejä tarvitaan laskennan suorittamiseen.

Kineettisten laskentojen toteutuksen viimeisenä vaiheena oli testien liittäminen osaksi Settings-tiedostoa.

```
- <class name="KineticTest">
  - <method name="setup">
      <System.String>c:\Moduletest\xmldata\BasicStatKinetifluoro.xml</System.String>
    </method>
    <method name="KineticAvgRateS" />
    <method name="KineticMaxRateU" />
    <method name="KineticTimeMaxRate" />
    <method name="KineticTimeToChange" />
    <method name="KineticMin" />
    <method name="KineticMax" />
    <method name="KineticSum" />
    <method name="KineticIntegral" />
  - <method name="setup">
      <System.String>c:\Moduletest\xmldata\BasicStatKineticPhoto.xml</System.String>
    </method>
    <method name="KineticAvgRateS" />
    <method name="KineticMaxRateU" />
    <method name="KineticTimeMaxRate" />
    <method name="KineticTimeToChange" />
    <method name="KineticMin" />
    <method name="KineticMax" />
    <method name="KineticSum" />
    <method name="KineticIntegral" />
  - <method name="setup">
      <System.String>c:\Moduletest\xmldata\basicfluoro.xml</System.String>
    </method>
    <method name="KineticAvgRateS" />
    <method name="KineticMaxRateU" />
    <method name="KineticTimeMaxRate" />
    <method name="KineticTimeToChange" />
    <method name="KineticMin" />
    <method name="KineticMax" />
    <method name="KineticSum" />
    <method name="KineticIntegral" />
  </class>
```

Kuva 25. Kineettisten testien liittäminen osaksi automatisointia

Kuva 25 esittää kuinka kineettisiä laskentoja testaavat testitapaukset näkyvät osana Settings-tiedostoa. Kuten kuvasta näkyy, Setup-funktiota kutsutaan kolme kertaa. Kahdella ensimmäisellä Setup-funktion kutsukerralla sille välitetään kineettisillä mittausteknologilla suoritettuja mittauksia ja viimeisellä kutsukerralla välitetään parametrina sellaisia

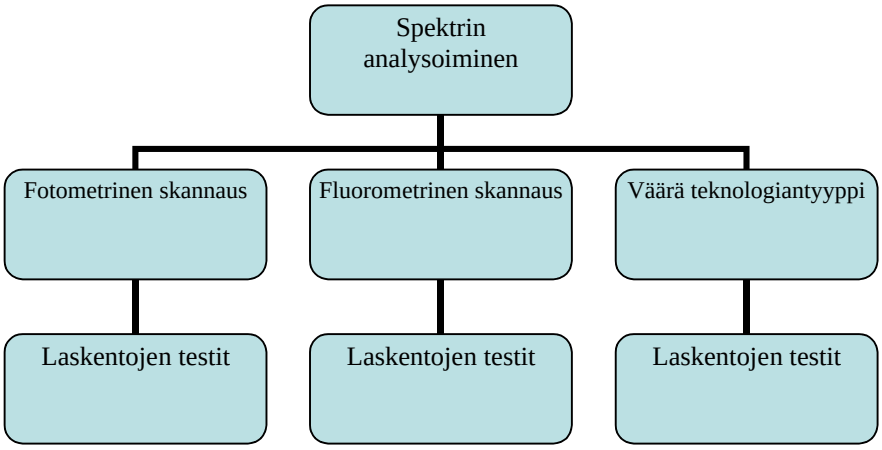
mittaustuloksia, joita ei ole suoritettu kineettisesti. Näin saadaan kaikki kuvan 23 haarat käytyä läpi.

8.7 Spektrin tulkinta

Luvussa käsitellään spektrin analysoinnissa käytettyjen laskentojen testaamista ja suunnittelua. Laskennat esiteltiin luvussa 8.7.

8.7.1 Spektrin tulkintatestauksen suunnittelu

Laskentoja pystyy suorittamaan mittaustuloksista ainoastaan silloin, kun mittaus on suoritettu joko fluoro- tai fotometrisenä skannauksena. Jos teknologiatyyppinä on mittaus, laskennat eivät onnistu. Spektrin analysointiin liittyvät laskennat koostuvat 8 erilaisesta laskentatyypistä, kuten luvussa 8.7 kerrottiin.



Kuva 26. Spektrin analysointitestien hierarkia

Kuvassa 26 on johdettu määrittelyjen pohjalta testitapausten hierarkiakaavio. Kun kuvan 26 hierarkiamallin mukaisesti testataan spektrin analysointilaskentoja, pitää kaikkien muiden testien tuottaa kelvollisia tuloksia paitsi väärän teknologiatyypin alla olevien testien. Nämä testit eivät saa tuottaa minkäänlaisia tuloksia. Kun on valittu väärä teknologiatyyppi, laskentojen suorittamisen pitää loppua tiedon tarkistamisvaiheessa.

8.7.2 Spektrin tulkintatestauksen toteutus

Laskentojen testaamista varten pitää tuottaa kolmet eri mittaustulokset. Tarvitaan siis Fotometrinen ja Fluorometrinen skannaus. Viimeinen mittaus suoritetaan jollakin väärällä teknologiatyypillä.

Ensimmäinen tehtävä oli luoda laskennalle oma testiohjelma. Testiohjelman kuvaus esiteltiin luvussa 8.2.2. Seuraavaksi luotiin Setup-funktio testiohjelmaan. Setup-funktion kuvaus esiteltiin kappaleessa 8.2.3

Jokaiselle spektrin analysointilaskennalle piti luoda oma testifunktio. Funktioiden prototyypit ovat seuraavanlaiset:

- public double FindPeaks() spektrissä esiintyvän piikin etsimisen testaamiseen
- public double FindMaxRate() suurimman nousunopeuden etsimisen testaamiseen
- public double FindTimeToMaxrate() suurimman nousunopeuden ajankohdan selvittämisen testaamiseen
- public double FindTimeToChange halutun muutoksen tapahtumiseen kuluvan ajan laskemisen testaamiseen
- public double FindMin pienimmän arvon etsimisen testaamiseen
- public double FindMax suurimman arvon etsimisen testaamiseen ja
- public double FindRatioWithInSpetrum kahden eri spektrin suhdeluvun testaamiseen.

Toteutukseltaan nämä funktiot ovat äärimmäisen yksinkertaisia. Niissä kutsutaan Calculate-luokan ilmentymän avulla ICalculate-rajapinnassa olevaa Calculate-funktiota. Calculate-funktio toimittaa Spektrin analysointilaskentoja toteuttavalle luokalla mittaustulokset ja parametrit. TestReflector löytää syntyneet tulokset ResultItem-luokasta ja tekee näiden tietojen perusteella päätöksen, onko testi onnistunut vai ei.

Spektrin analysointitestien viimeisenä vaiheena oli liittää testit osaksi Settings-tiedostoa. Kuva 27 esittää sen, miten yksittäiset testifunktiot näkyvät osana testattavaa kokonaisuutta.

```
- <class name="ProcessbaseTest">
    <method name="setup" />
    <System.String>c:\Moduletest\xmldata\fotoScanning.xml</System.String>
    <method name="FindPeaks" />
    <method name="FindMaxRate" />
    <method name="FindTimeTOMaxRate" />
    <method name="FindMax" />
    <method name="FindMin" />
    <method name="FindHBW" />
    <method name="FindTimeToChange" />
    <method name="setup" />
    <System.String>c:\Moduletest\xmldata\FluoroScanning.xml</System.String>
    <method name="FindPeaks" />
    <method name="FindMaxRate" />
    <method name="FindTimeTOMaxRate" />
    <method name="FindMax" />
    <method name="FindMin" />
    <method name="FindHBW" />
    <method name="FindTimeToChange" />
  </class>
```

Kuva 27. Spektrin analysointien testien liittäminen osaksi testauksen automatisointia

Kuvassa 27 esittää, kuinka kineettiset laskentojen testitapaukset näkyvät osana Settings-tiedostoa. Kuten kuvasta näkyy, Setup-funktiota kutsutaan kaksi kertaa. Kummallakin Setup-funktion kutsukerralla sille välitetään eri mittausteknologialla suoritettuja mittauksia. Näin saadaan käytyä kaikki kuvan 26 haarat läpi.

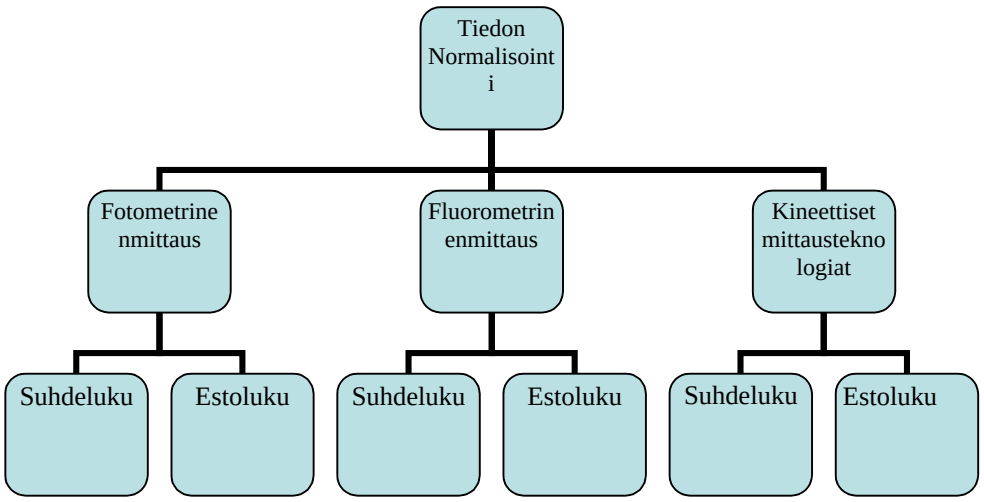
8.8 Tiedon normalisointitestit

Tässä luvussa esitellään tiedon normalisointitestien suunnittelua ja toteuttamista. Itse laskennat esiteltiin kappaleessa 6.3.

8.8.1 Tiedon normalisointitestien suunnittelu

Laskentoja toteuttava luokka koostuu kolmesta funktiosta. Prosescal-funktion tehtävä on tarkistaa, että luokalle on välitetty tarvittavat parametrit, joita vaaditaan laskennan suorittamiseen. ProcessCal-funktion tehtävänä on myös selvittää annetuista parametreista, onko kyseessä suhdeluku- vai estolukulaskenta. Varsinaisen laskentojen toteuttamiseen tarvitaan kaksi funktiota. Suhdelukulaskentaa toteutetaan Ratio-funktiossa ja estoluku-laskentaa suoritetaan Inhibation-funktiossa.

Tiedon normalisoinnin pitää toimia kaikilla mittausteknologioilla ja teknologiatyypeillä. Kineettisyys ei saa vaikuttaa laskennan suorittamiseen.



Kuva 28. Suhdeluku- ja estolukulaskentojen testitapaukset

Kuvassa 28 on johdettu määrittelyjen pohjalta testitapausten hierarkiakaavio. Kuvan 28 kaikista haaroista pitää syntyä tuloksia.

8.8.2 Tiedon normalisointitestien toteutus

Tiedon normalisointilaskentojen testien toteutus aloitettiin luomalla luvun 8.2.2 tavalla oma testiympäristö. Seuraavaksi tarvittiin mittaustiedon käsittelyä ja tietorakenteiden alustamista varten oma Setup-funktio. Setup-funktion toteutus esiteltiin luvussa 8.2.3. Lisäksi tarvitaan kolmella eri mittausteknologialla mittaustuloksia testausta varten kuvan 25 mukaisesti.

Testiohjelmassa toteutettiin Setup-funktion lisäksi suhdeluvun ja estoluvun testaamista varten omat funktionsa. Testifunktiot prototyypit ovat

- public double GetRatio() suhdeluvun testaamista varten
- public double GetInhibitio() estoluvun testaamista varten.

Funktioissa välitetään TestFormula-muuttujan avulla tieto Calculate-luokan Calculate-funktiolle. TestFormula-muuttujan sisältämän tiedon perusteella Calculate-funktio tekee päätöksen, kumpi laskenta on kyseessä, suhdeluku vai estoluku.

Viimeisenä testien toteutuksen vaiheena testit liitettiin osaksi automatisoitavaa kokonaisuutta.

```
- <class name="RatioInhibitioTest">
  - <method name="SetUp">
      <System.String>c:\Moduletest\xmldata\Fluoro.xml</System.String>
    </method>
    <method name="RatioTest" />
    <method name="InhibitionTest" />
  - <method name="SetUp">
      <System.String>c:\Moduletest\xmldata\Photo.xml</System.String>
    </method>
    <method name="RatioTest" />
    <method name="InhibitionTest" />
  - <method name="SetUp">
      <System.String>c:\Moduletest\xmldata\KineticPhoto.xml</System.String>
    </method>
    <method name="RatioTest" />
    <method name="InhibitionTest" />
  </class>
```

Kuva 29. Tiedon normalisointitestien liittäminen osaksi testauksen automatisointia

Kuva 29 esittää, miten tiedon normalisointitestit näkyvät Settings-tiedossa. Jotta saadaan suoritettua kuvan 28 testihierarkiakaavion kaikki haarat, pitää Setup-funktiota kutsua kolme kertaa ja välittää sille eri mittausteknologioilla suoritetut mittaukset.

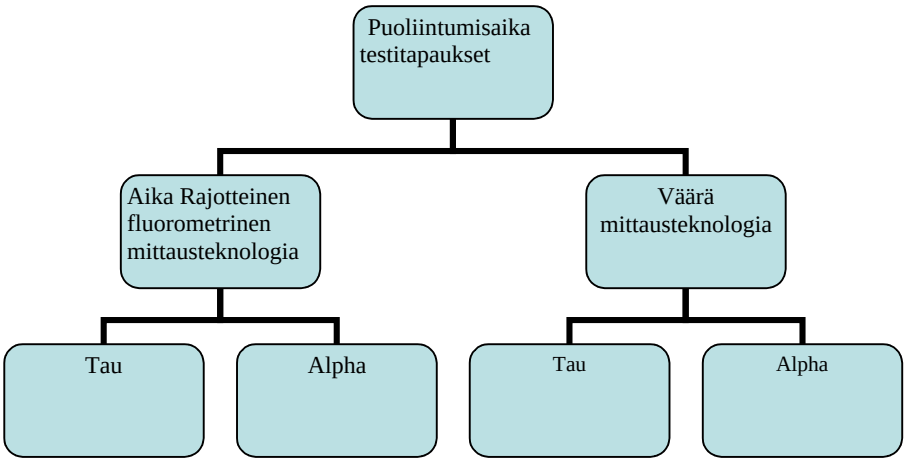
8.9 Puoliintumisajan testaaminen

Tässä luvussa esitellään käyrän puoliintumisaikaan (engl. decay calculation test) liittyvien testien suunnittelua ja toteuttamista. Itse laskennat esiteltiin kappaleessa 6.6.

8.9.1 Puoliintumisajantestien suunnittelu

Laskentoja toteuttava luokka koostuu funktioista ProcessCall ja Calculate. ProcessCal-funktion tehtävä on tarkistaa, että laskennalle välitettävä ResultContainer-luokan ilmentymän sisältämä tieto on oikeanlaista. Calculate-funktion tehtävänä on laskea Taun ja Alphan arvot.

Puoliintumisaikalaskennat liittyvät ainoastaan aikaerotteiseen fluorimetriseen mittaukseen. Laskennat eivät saa tuottaa minkäänlaisia tuloksia millään muulla mittausteknologialla kuin aikaerotteisella fluorimetrisellä mittauksella. Näistä vaatimuksista voidaan johtaa seuraavanlainen testitapauskaavio.



Kuva 30. Puoliintumisaikatestitapausten hierarkia

Kuvassa 30 testitapaukset on jaettu niin, että kuvan vasemman puolen tapaukset on suoritettu oikealla mittausteknologialla, jolloin kaikki laskentaa liittyvät parametrit ovat juuri oikeanlaiset. Kun kuvan 30 oikea puolisko on suoritettu väärällä mittausteknologilla, mittauksessa syntyneet parametrit ovat vääränlaiset laskennan suorittamista varten. Kun kyseessä on väärä mittausteknologia, laskentojen suorittamisen pitää keskeytyä eikä minkäänlaisia tuloksia saa syntyä.

8.9.2 Puoliintumisaikatestien toteutus

Puoliintumisaikatestien toteuttaminen aloitettiin luomalla kappaleen 8.2.2 tavalla oma testiympäristö. Seuraavaksi tarvittiin mittaustiedon käsittelyä ja tietorakenteiden alustamista varten oma Setup-funktio. Setup-funktion toteutus esiteltiin kappaleessa 8.2.3.

Kummallekin Tau- ja Alpha-laskennoille tarvittiin oma testi-funktionsa. Tauta ja Alphaa testaavien funktioiden prototyypit olivat seuraavanlaiset:

- public Double GetTau(), Taun tuloksia varten
- public Double GetAlpha() Alphan tuloksia varten

Testien suorittamista varten tarvittiin kaksi sarjaa erilaisia mittaustuloksia. Yksi sarja mittaustuloksia, joka oli suoritettu oikealla mittausteknologialla ja toinen joka oli suoritettu väärällä mittausteknologialla. Kaikki mittaustulokset olivat XML-tiedostona, kuten kappaleessa 8.1.3 todettiin. Mittaustuloksia sisältää XML-tiedosto sisältää myös tiedon kyseessä olevasta mittausteknologiasta.

Kun testit oli tehty ja niiden toteutus oli todettu toimivaksi, piti puoliintumisaikatestit liittää osaksi automatisoitavaa kokonaisuutta. Kuva 31 esittää, miten testit näkyivät osana Settings-tiedostoa.

```
- <class name="DecayCalculationTest">
  - <method name="Setup">
      <System.String>c:\Moduletest\xmldata\Decay.xml</System.String
    </method>
    <method name="GetTau" />
    <method name="GetTAlpha" />
  - <method name="Setup">
      <System.String>c:\Moduletest\xmldata\FOTO.xml</System.String>
    </method>
    <method name="GetTau" />
    <method name="GetTAlpha" />
  </class>
```

Kuva 31. Puoliintumisaikatestien liittäminen osaksi automatisointia

Jotta saadaan suoritettua kuvan 30 kaikki testihierarkiakaavion haarat, pitää Setup-funktiota kutsua kaksi kertaa. Setup-funktion ensimmäisellä kutsukerralla välitetään oikealla mittausteknologialla suoritetut mittaustulokset ja toisella kerralla väärällä mittausteknologialla suoritetut mittaukset.

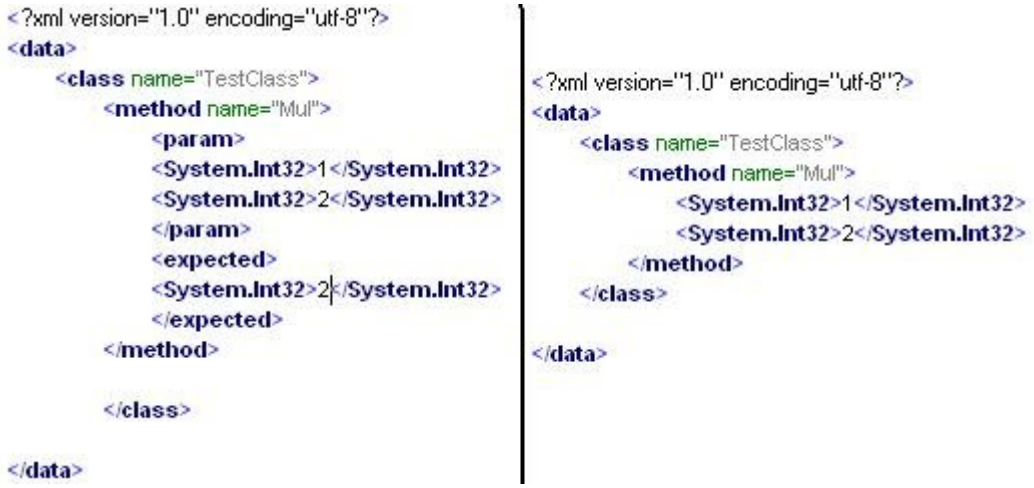
9 KEHITYSEHDOTUKSIA

Jos yrityksessä halutaan kehittää ohjelmiston testauksen automatisointia, kannattaa miettiä asiaa ensiksi työkalun näkökulmasta. Työssä käytettiin TestReflector-nimistä ohjelmaa. Ohjelmassa on potentiaalia, mutta tämänhetkistä ohjelmaversiota vaivaa muutama käytettävyysongelma ja sen arkkitehtuuriin liittyviä ongelmia.

Suurin hienous mitä TestReflector pitää sisällään on menetelmätapa, jolla voidaan ottaa käyttöön luokkien funktiot riippumatta niiden näkyvyysmääreestä. Testaajan tarvitsee ainoastaan tietää missä dll-tiedostossa ja luokassa funktio sijaitsee testatakseen funktion toiminnan.

Käytettävyysongelmista pahin liittyy päätöksentekomenettelyyn. TestReflector tekee päätöksen testin onnistumisesta Expected- ja Actual-lokitiedoston perusteella. Ohjelmisto vertailee lokitiedostojen rivejä keskenään. Jos ne eivät ole samat, niin testi epäonnistuu. Yksikin ylimääräinen välilyönti aiheuttaa testin epäonnistumisen. Ylimääräisiä rivejä syntyy yllättävän helposti, jos jostakin syystä pitää editoida Expected-lokitiedostoa.

TestReflectorin arkkitehtuuriin liittyy yksi epäkohta. TestReflector vaatii toimiakseen XML-tiedoston, joka sisältää testauksen ohjeet ja yhden lokitiedoston, jossa on odotettavissa olevat tulokset. Miksi tällaiseen ratkaisuun on päädytty? Jos TestReflectoria halutaan tulevaisuudessa kehittää, hyvä idea kehityssuunnalle olisi sellainen, että poistetaan kokonaan Expected-lokitiedosto, ja sen sisältämä informaatio siirretään XML-tiedostoon, joka sisältää testauksen ohjeet. Jokaisen testifunktion kohdalle lisättäisiin yksi ylimääräinen XML-tunniste esimerkiksi kuvan 34 osoittamalla tavalla.



Kuva 32. Kehitysidea Settings-tiedostosta

Kuvan 32 oikea puoli kuvaa mitä TestReflectorin Settings-tiedosto on tällä hetkellä ja vasen puoli mitä se voisi olla. Kuvan ideana on se, että nykyisen TestReflectorin Settings-tiedoston avulla kerrotaan testattavan luokan nimi, mitä luokan metodeja testataan ja millä parametreillä. Tulevaisuudessa settings-tiedoston avulla voitaisiin myös kertoa, minkä tuloksen funktion pitää palauttaa. Kuvan 32 esimerkissä odotettavissa oleva tulos löytyy Expected XML-tunnisteen sisältä. Tällä tavalla päästäisiin eroon Expected-lokitiedoston liittyvistä ongelmista Tämä valitettavasti vaati aika suurta muutosta ohjelman rakenteessa ja se sitoo resursseja pois ”tuottavasta työstä”.
Jos TestReflectorin kehittämistä ei nähdä hyvänä vaihtoehtona, hyviä TestReflectorin korvaavia ilmaisia ohjelmia ovat mm. Nunit ja Testdriven. Maksullisista ohjelmista parhaimmistoon kuuluu Microsoftin Kehittämät Visual Studio Team System for testers tai ACL.

Testauksen automatisointia ei missään nimessä kannata lopettaa tähän, kun se on kerran aloitettu tämän työn merkeissä. Testauksen automatisoinnin hyödyntämistä kannattaa miettiä myös muistakin ohjelman osa-alueista. Hyviä testauksen automatisoinnin kohteita voisi löytyä esimerkiksi ARB-testien joukosta. Muita hyviä kohteita voisi löytyä esimerkiksi kuvan 4 kaavion ryhmän muut alta. Ryhmä muut piti sisällään hyvin sekalaisia testitapauksia, joita ei voinut ryhmitellä mihinkään kategoriaan. Sieltä voisi hyvinkin löytyä luvussa 5.3 esitettyjen kriteerien valossa sellaisia testitapauksia, joiden automatisointi olisi mielekästä.

Lopuksi on vielä yksi ohjelmiston automatisoinnin kannalta mielenkiintoinen asia. Työssä alustettiin kaikki laskentoihin tarvittavat tietorakenteet XML-dokumentissa olevilla mittaustuloksilla ja mittausparametreilla. Tämä XML-

dokumentti on tuotu (engl. export) ohjelmiston käyttämästä tietokannasta. Kannattaa miettiä tulevaisuudessa, käytetäänkö tämän tyylistä ratkaisua vai haetaanko kaikki tieto, mitä tarvitaan, ohjelmiston tietorakenteiden alustamiseen suoraan ohjelmiston käyttämästä tietokannasta. Tällä tavalla saavutetaan kolme merkittävää etua. Päästään eroon XML-dokumenttien käsittelystä lähes kokonaan. Toiseksi testien ylläpidettävyys helpottuu huomattavasti, kun testiarvojen hakutietokannasta käytetään yksilöllisiä id-numeroita. Kolmantena seikkana laskentojen testauksen yhteydessä tulee testattua myös ohjelmiston logiikka, jolla se alustaa omat tietorakenteensa.

10 YHTEENVETO

Tässä opinnäytetyössä käytiin läpi ohjelmistotestauksen automatisoinnin mahdollisuuksia. Työn alkupuoli keskittyi testauksen ja testauksen automatisoinnin teorian käsittelyyn. Insinöörityön pääpaino on käytännössä. Suurin osa työstä on käytetty potentiaalisten automatisoitavien testikohteiden etsimiseen ja automatisoinnin toteutukseen ja suunnitteluun.

Tarkoituksena ei ole ollut käsitellä kaikkia testaamiseen tai automatisointiin liittyviä teorioita, vaan ainoastaan niitä jotka ovat työn kannalta tärkeitä. Kuten edellä jo mainittiin, työn tärkein osa on testauksen automatisoinnin toteutus.

Varsinainen käytännön osuus aloitettiin esittelemällä yrityksen käyttämiä menetelmiä ohjelmistojen kehityksessä ja testauksessa. Ennen varsinaisten automatisoitavien kohteiden etsimistyön aloittamista, määriteltiin neljä kriteeriä. Testitapausten piti täyttää nämä neljä kriteeriä, jotta automatisointiin edes ryhdyttiin. Seuraava vaihe automatisointikohteiden etsimisessä oli se, että käytiin läpi nykyistä manuaalista testausjärjestelmää ja selvitettiin mihin aika testauksessa kului. Kun saatiin tarkka käsitys mihin aika testauksessa kuluu ja oli määritelty testitapauksilta vaadittavat kelpoisuuskriteerit, ryhdyttiin valitsemaan automatisoitavia kohteita. Automatisoitavaksi kohteeksi valittiin ohjelmistojen laskentojen testaus.

Testauksessa käytettävänä työkaluna oli TestReflector. Työkalu esiteltiin luvussa 7. Hieman erikoisen työkalun valinnan taustalla on se, että se on yrityksen itsensä kehittämä. Haluttiin selvittää onko TestReflectorista oikeasti ainesta varteenotettavaksi ohjelmiston testaustyökaluksi. TestReflectoria vaivaa vielä keskeneräisyyden vuoksi muutama pieni käytettävyyteen liittyvä ongelma, mutta pienellä kehitystyöllä siitä saa hyvän ja varsin toimivan testaustyökalun.

Testauksen suunnittelussa käytettiin apuna tietyn laskennan toteuttavaa koodia ja yrityksestä löytyvää laskennan testaukseen tarkoitettua manuaalista testaussuunnitelmaa. Näiden kahden apuvälineen avulla pyrittiin testaamaan koneellisesti laskennat mahdollisimman kattavasti. Jokaiselle laskennalle kirjoitettiin oma testiohjelma. Testiohjelman tärkeimmät tehtävät olivat käsitellä XML-tiedostoja, joissa oli kaikki

laskentojen suorittamiseen vaadittava tieto ja XML-tiedostosta saatavien tietojen pohjalta alustaa kaikki laskentojen vaatimat tietorakenteet. Testiohjelma kutsui määriteltyä rajapintaa, jonka kautta varsinaisia laskentoja suoritettiin. Lopuksi jokainen testiohjelma liitettiin osaksi koko laskentojen testausta automatisoivaa Settings-tiedostoa.

Ohjelmiston laskennat ovat tällä hetkellä hyvin stabiilissa tilassa ja moneen kertaan testattu manuaalisen testausjärjestelmän avulla. Testauksen automatisoinnin avulla ei löytynyt uusia virheitä. Suurin hyöty työn tekemisessä on se, että se vähentää testaajien työtaakkaa, kun heidän ei tarvitse testata manuaalisesti enää näitä asioita, vaan ne voidaan hoitaa automaattisesti. Työssä kehitetty laskentatestien automatisointijärjestelmä testaa yksittäisen laskennan toimivuuden, mutta ei testaa eri laskentojen yhdistämisestä syntyvän kombinaation syntyvän tuloksen oikeellisuutta.

VIITELUETTELO

[1] Pohjolainen Pentti, Ohjelmistotestauksen Automatisointi[www-dokumentti viitattu12.10.2006].
Saatavilla: http://www.cs.uku.fi/tutkimus/Teho/PenttiPohjolainen_Gradu.pdf.

[2] Haikala Ilkka& Märijärvi Jukka, Ohjelmistotuotanto. Talentum media, kymmenes painos 2004.

[3] Fester Mark, Graham Dorothy, Software Test Automation: effective use of the test execution tools, ACM Press books1999.

[4] Yritys, Tuote 1 usermanual,2006.

[5] Yritys, Tuote 2 usermanual,2006.

[6] Edward Kitt, Software testing in the real world, ACM Press book 1998.

[7] Kinni Toni, Ohjelmistotestauksen automatisointi ja sen soveltaminen puunhankinnan tietojärjestelmään[www-dokumentti Viitattu 10.11.2006].
Saatavilla: http://www.scp.fi/publications/c-sarja/isbn9525155803.pdf

[8] Salkola Mika, Suullinen tiedonanto.

[9] Yritys, Tuote 2 Test Specification, 2000

[10] Team Foundation Server [www-dokumentti viitattu 4.1.2007.]
Saatavilla: http://msdn2.microsoft.com/en-us/teamsystem/aa718916.aspx.

Settings-Tiedosto:

```
<?xml version="1.0" encoding="utf-8"?>
<data>
	<class name="SimpleCalculationTest">
		<method name="SimpleAvg"></method>
		<method name="SimpleSum"></method>
		<method name="SimpleMinValue"></method>
		<method name="SimpleMaxValue"></method>
		<method name="SimpleMaxMinusMin"></method>
	</class>
	<class name="DecayCalculationTest">
		<method name="Setup">
			<System.String>c:\Moduletest\xmldata\Decay.xml</System.String>
		</method>
		<method name="GetTau"></method>
		<method name="GetTAlpha"></method>
	</class>
	<class name="BlankTest">
		<method name="Setup">
			<System.String>c:\Moduletest\xmldata\BlankFoto.xml</System.String>
		</method>
		<method name="BlankTestAvg"></method>
		<method name="BlankTestSpecifig"></method>
		<method name="Setup">
			<System.String>c:\Moduletest\xmldata\Blankfluro.xml</System.String>
		</method>
		<method name="BlankTestAvg"></method>
		<method name="BlankTestSpecifig"></method>
		<method name="Setup">
			<System.String>c:\Moduletest\xmldata\BlankNoBlank.xml</System.String>
				</method>
	</class>
	<class name="BasicStaticTest">
		<method name="Setup">
			<System.String>c:\Moduletest\xmldata\BasicStatFoto.xml</System.String>
		</method>
		<method name="GetCV"></method>
		<method name="GetSD"></method>
		<method name="GetAvg"></method>
		<method name="Setup">
			<System.String>c:\Moduletest\xmldata\BasicStatfluoro.xml</System.String>
		</method>
		<method name="GetCV"></method>
		<method name="GetSD"></method>
		<method name="GetAvg"></method>
		<method name="Setup">
			<System.String>c:\Moduletest\xmldata\BasicStatKineticPhoto.xml</System.String>
				</method>
		<method name="GetCV"></method>
		<method name="GetSD"></method>
		<method name="GetAvg"></method>
		<method name="Setup">
			<System.String>c:\Moduletest\xmldata\BasicStatKineticFluro.xml</System.String>
		</method>
		<method name="GetCV"></method>
		<method name="GetSD"></method>
		<method name="GetAvg"></method>
	</class>
```

```
<class name="KineticTest">
    <method name="setup">
        <System.String>c:\Moduletest\xmldata\BasicStatKinetifluoro.xml</System.String>
    </method>
    <method name="KineticAvgRateS"></method>
    <method name="KineticMaxRateU"></method>
    <method name="KineticTimeMaxRate"></method>
    <method name="KineticTimeToChange"></method>
    <method name="KineticMin"></method>
    <method name="KineticMax"></method>
    <method name="KineticSum"></method>
    <method name="KineticIntegral"></method>
    <method name="setup">
        <System.String>c:\Moduletest\xmldata\BasicStatKineticPhoto.xml</System.String>
    </method>
    <method name="KineticAvgRateS"></method>
    <method name="KineticMaxRateU"></method>
    <method name="KineticTimeMaxRate"></method>
    <method name="KineticTimeToChange"></method>
    <method name="KineticMin"></method>
    <method name="KineticMax"></method>
    <method name="KineticSum"></method>
    <method name="KineticIntegral"></method>
</class>
<class name="RatioInhibitioTest">
    <method name="SetUp">
        <System.String>c:\Moduletest\xmldata\BasicStatKineticPhoto.xml</System.String>
    </method>
    <method name="RatioTest"></method>
    <method name="InhibitionTest"></method>
    <method name="SetUp">
        <System.String>c:\Moduletest\xmldata\BasicStatKineticPhoto.xml</System.String>
    </method>
    <method name="RatioTest"></method>
    <method name="InhibitionTest"></method>
</class>
<class name="ProcessbaseTest">
    <method name="setup">
        <System.String>c:\Moduletest\xmldata\fotoScanning.xml</System.String>
    </method>
    <method name="FindPeaks"></method>
    <method name="FindMaxRate"></method>
    <method name="FindTimeTOMaxRate"></method>
    <method name="FindMax"></method>
    <method name="FindMin"></method>
    <method name="FindHBW"></method>
    <method name="FindTimeToChange"></method>
    <method name="setup"></method>
        <System.String>c:\Moduletest\xmldata\FluoroScanning.xml</System.String>
    <method name="FindPeaks"></method>
    <method name="FindMaxRate"></method>
    <method name="FindTimeTOMaxRate"></method>
    <method name="FindMax"></method>
    <method name="FindMin"></method>
    <method name="FindHBW"></method>
    <method name="FindTimeToChange"></method>
</class>
```